数据仓库服务(DWS)
8.1.3
SQL 语法参考

文档版本 11

发布日期 2024-05-08

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 GaussDB(DWS) SQL 概述

什么是 SQL

SQL是用于访问和处理数据库的标准计算机语言。

SQL提供了各种任务的语句，包括：

- 查询数据。
- 在表中插入，更新和删除行。
- 创建，替换，更改和删除对象。
- 控制对数据库及其对象的访问。
- 保证数据库的一致性和完整性。

SQL语言由用于处理数据库和数据库对象的命令和函数组成。该语言还会强制实施有关数据类型、表达式和文本使用的规则。因此在SQL参考章节，除了SQL语法参考外，还会看到有关数据类型、表达式、函数和操作符等信息。

SQL 发展简史

SQL发展简史如下：

- 1986年，ANSI X3.135-1986，ISO/IEC 9075:1986，SQL-86
- 1989年，ANSI X3.135-1989，ISO/IEC 9075:1989，SQL-89
- 1992年，ANSI X3.135-1992，ISO/IEC 9075:1992，SQL-92（SQL2）
- 1999年，ISO/IEC 9075:1999，SQL:1999（SQL3）
- 2003年，ISO/IEC 9075:2003，SQL:2003（SQL4）
- 2011年，ISO/IEC 9075:200N，SQL:2011（SQL5）

GaussDB(DWS)支持的 SQL 标准

GaussDB(DWS)兼容Postgres-XC，默认支持SQL2、SQL3和SQL4的主要特性、SQL5的部分特性。

GaussDB(DWS)支持的语言扩展

GaussDB(DWS)支持的语言扩展扩展包括PL/pgSQL、PL/Java、PL/R。

SQL 语法文本格式约定

为了方便对语法使用的理解，在文档中对SQL语法文本按如下格式进行表述。

格式	意义
大写	语法关键字（语句中保持不变、必须与语法格式一致的部分）采用大写表示。
小写	参数（语句中必须由实际值进行替代的部分）采用小写表示。
[ ]	可选语法项。表示用“[ ]”括起来的部分是可选的。
{ }	必选语法项。
...	表示前面的元素可重复出现。
[x | y | ...]	表示从两个或多个选项中选取一个或者不选。
{ x | y | ... }	表示从两个或多个选项中选取一个。
[x | y | ...] [...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用空格分隔。
[x | y | ...] [,...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用逗号分隔。
{ x | y | ... } [...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间以空格分隔。
{ x | y | ... } [,...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间用逗号分隔。

SQL 示例说明

手册中的部分SQL示例是基于TPC-DS模型开发的，如果需要运行手册中的示例，请先参考官网说明（**http://www.tpc.org/tpcds/**），安装TPC-DS benchmark。

2 与 PostgreSQL 的差异

GaussDB(DWS)与PostgreSQL的差异基于PostgreSQL 9.X版本整理，具体差异如下：

客户端差异

GaussDB(DWS) gsql相较于PostgreSQL psql做了如下安全加固变更：

- 取消通过元命令\password设置用户密码。
- 新增元命令\i+、 \ir+、\include_relative+和输入输出参数-k，以支持给导入导出的文件加密。
- 取消打印命令行历史到文件的元命令\s。
- 涉及敏感操作SQL历史不再记录，如含有密码操作。即用户通过翻页/上下键查阅SQL历史将不能查到对应的记录。
- 支持连接后在屏幕上给出用户密码过期提示以及版本信息。

gsql在psql基础上还增加了如下功能：

- 新增输出格式参数-r。支持用户输入命令时的tab补齐和方向键调整焦点。
- 新增并行操作元命令\parallel，以提升执行性能。
- 新增\set RETRY支持语句出错重试。
- 新增PLSQL默认结束符功能，将“/”作为PLSQL语句（ create or replace function/procedure)的默认结束符，增加便利性。

libpq：

GaussDB(DWS)在开发某些功能，如客户端连接工具gsql时，对PostgreSQL libpq进行了较大修改，但并未对此接口在应用程序开发场景下的使用做验证。因此对使用此接口做应用程序开发存在的风险未知，故不推荐用户使用此套接口做应用程序开发。推荐用户使用ODBC或JDBC接口来替代。

SQL 差异

表 2-1 GaussDB(DWS)不支持的 PostgreSQL 数据库语言

分类	GaussDB(DWS)不支持	说明
数据类型	几何类型line	GaussDB(DWS)所支持的数据类型参见**数据类型**。
	pg_node_tree	
函数	枚举支持函数： • enum_first(anyenum) • enum_last(anyenum) • enum_range(anyenum) • enum_range(anyenum, anyenum)	GaussDB(DWS)所支持的函数参见**函数和操作符**。
	访问权限查询函数： • has_sequence_privilege(user, sequence, privilege) • has_sequence_privilege(sequence, privilege)	
	系统目录信息函数： • pg_get_triggerdef(trigger_oid) • pg_get_triggerdef(trigger_oid, pretty_bool)	
	几何类型转换函数： line(point, point)	
	pg_node_tree函数	
SQL语法	CREATE TABLE子句： INHERITS (parent_table [, ...])	继承表。
	CREATE TABLE的列约束： REFERENCES reftable [(refcolumn)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action] [ON UPDATE action]	列约束中用REFERENCES reftable [(refcolumn)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action] [ON UPDATE action] 为表创建外键约束。
	CREATE TABLE的表约束： EXCLUDE [USING index_method] (exclude_element WITH operator [, ...])	表约束中用EXCLUDE [USING index_method] (exclude_element WITH operator [, ...])为表创建排除约束。
	CREATE/ALTER/DROP EXTENSION	扩展的加载、修改和删除。

分类	GaussDB(DWS)不支持	说明
	CREATE/ALTER/DROP AGGREGATE	聚集函数的定义、修改和删除。
	CREATE/ALTER/DROP OPERATOR	操作符（OPERATOR）的创建、修改和删除。
	CREATE/ALTER/DROP OPERATOR CLASS	操作符类（OPERATOR CLASS）的创建、修改和删除。
	CREATE/ALTER/DROP OPERATOR FAMILY	操作符族（OPERATOR FAMILY）的创建、修改和删除。
	CREATE/ALTER/DROP TEXT SEARCH PARSER	文本检索解析器（TEXT SEARCH PARSER）的创建、修改和删除。
	CREATE/ALTER/DROP TEXT SEARCH TEMPLATE	文本检索模板（TEXT SEARCH TEMPLATE）的创建、修改和删除。
	CREATE/ALTER/DROP COLLATION	排序规则（COLLATION）的创建、修改和删除
	CREATE/ALTER/DROP CONVERSION	字符集编码转换（CONVERSION）的定义、修改和删除。
	CREATE/ALTER/DROP RULE	规则（RULE）的创建、修改和删除。
	CREATE/ALTER/DROP LANGUAGE	过程语言（LANGUAGE）的注册、修改和删除。
	CREATE/ALTER/DROP DOMAIN	域（DOMAIN）的创建、修改和删除。
	CREATE/DROP CAST	类型转换（CAST）的定义和删除。
	CREATE/ALTER/DROP USER MAPPING	用户映射（USER MAPPING）的定义、修改和删除。
	SECURITY LABEL	定义或更改对象的安全标签。
	NOTIFY	生成一个通知。
	LISTEN	监听一个通知。
	UNLISTEN	停止监听通知信息。
	LOAD	加载或重新加载一个共享库文件。

分类	GaussDB(DWS)不支持	说明
	DISCARD	释放一个数据库的会话资源。（8.2.0及以上集群版本已支持DISCARD。）
	MOVE BACKWARD	反向移动游标。
	COPY的COPY FROM FILE和COPY TO FILE	为了权限的隔离，GaussDB(DWS)禁用COPY FROM FILE和COPY TO FILE。
其他	自定义C函数	DWS支持的用户自定义函数参见**用户自定义函数**。

3 关键字

SQL里有保留字和非保留字之分。根据标准，保留字绝不能用做其他标识符。非保留字只是在特定的环境里有特殊的含义，而在其他环境里是可以用做标识符的。

表 3-1 SQL 关键字

关键字	GaussDB(DWS)	SQL:1999	SQL-92
ABORT	非保留	-	-
ABS	-	非保留	-
ABSOLUTE	非保留	保留	保留
ACCESS	非保留	-	-
ACCOUNT	非保留	-	-
ACTION	非保留	保留	保留
ADA	-	非保留	非保留
ADD	非保留	保留	保留
ADMIN	非保留	保留	-
AFTER	非保留	保留	-
AGGREGATE	非保留	保留	-
ALIAS	-	保留	-
ALL	保留	保留	保留
ALLOCATE	-	保留	保留
ALSO	非保留	-	-
ALTER	非保留	保留	保留
ALWAYS	非保留	-	-
ANALYSE	保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
ANALYZE	保留	-	-
AND	保留	保留	保留
ANY	保留	保留	保留
APP	非保留	-	-
ARE	-	保留	保留
ARRAY	保留	保留	-
AS	保留	保留	保留
ASC	保留	保留	保留
ASENSITIVE	-	非保留	-
ASSERTION	非保留	保留	保留
ASSIGNMENT	非保留	非保留	-
ASYMMETRIC	保留	非保留	-
AT	非保留	保留	保留
ATOMIC	-	非保留	-
ATTRIBUTE	非保留	-	-
AUTHID	保留	-	-
AUTHINFO	非保留	-	-
AUTHORIZATION	保留(可以是函数或类型)	保留	保留
AUTOEXTEND	非保留	-	-
AUTOMAPPED	非保留	-	-
AVG	-	非保留	保留
BACKWARD	非保留	-	-
BARRIER	非保留	-	-
BEFORE	非保留	保留	-
BEGIN	非保留	保留	保留
BETWEEN	非保留(不能是函数或类型)	非保留	保留
BIGINT	非保留(不能是函数或类型)	-	-
BINARY	保留(可以是函数或类型)	保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
BINARY_DOUBLE	非保留(不能是函数或类型)	-	-
BINARY_INTEGER	非保留(不能是函数或类型)	-	-
BIT	非保留(不能是函数或类型)	保留	保留
BITVAR	-	非保留	-
BIT_LENGTH	-	非保留	保留
BLOB	非保留	保留	-
BOOLEAN	非保留(不能是函数或类型)	保留	-
BOTH	保留	保留	保留
BUCKETS	保留	-	-
BREADTH	-	保留	-
BY	非保留	保留	保留
C	-	非保留	非保留
CACHE	非保留	-	-
CALL	非保留	保留	-
CALLED	非保留	非保留	-
CARDINALITY	-	非保留	-
CASCADE	非保留	保留	保留
CASCADED	非保留	保留	保留
CASE	保留	保留	保留
CAST	保留	保留	保留
CATALOG	非保留	保留	保留
CATALOG_NAME	-	非保留	非保留
CHAIN	非保留	非保留	-
CHAR	非保留(不能是函数或类型)	保留	保留
CHARACTER	非保留(不能是函数或类型)	保留	保留
CHARACTERISTICS	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
CHARACTER_LENGTH	-	非保留	保留
CHARACTER_SET_CATALOG	-	非保留	非保留
CHARACTER_SET_NAME	-	非保留	非保留
CHARACTER_SET_SCHEMA	-	非保留	非保留
CHAR_LENGTH	-	非保留	保留
CHECK	保留	保留	保留
CHECKED	-	非保留	-
CHECKPOINT	非保留	-	-
CLASS	非保留	保留	-
CLEAN	非保留	-	-
CLASS_ORIGIN	-	非保留	非保留
CLOB	非保留	保留	-
CLOSE	非保留	保留	保留
CLUSTER	非保留	-	-
COALESCE	非保留(不能是函数或类型)	非保留	保留
COBOL	-	非保留	非保留
COLLATE	保留	保留	保留
COLLATION	保留(可以是函数或类型)	保留	保留
COLLATION_CATALOG	-	非保留	非保留
COLLATION_NAME	-	非保留	非保留
COLLATION_SCHEMA	-	非保留	非保留
COLUMN	保留	保留	保留
COLUMNS	非保留	-	-
COLUMN_NAME	-	非保留	非保留
COMMAND_FUNCTION	-	非保留	非保留

关键字	GaussDB(DWS)	SQL:1999	SQL-92
COMMAND_FUNCTION_CODE	-	非保留	-
COMMENT	非保留	-	-
COMMENTS	非保留	-	-
COMMIT	非保留	保留	保留
COMMITTED	非保留	非保留	非保留
COMPATIBLE_ILLEGAL_CHARS	非保留	-	-
COMPLETE	非保留	-	-
COMPRESS	非保留	-	-
COMPLETION	-	保留	-
CONCURRENTLY	保留(可以是函数或类型)	-	-
CONDITION	-	-	-
CONDITION_NUMBER	-	非保留	非保留
CONFIGURATION	非保留	-	-
CONNECT	-	保留	保留
CONNECTION	非保留	保留	保留
CONNECTION_NAME	-	非保留	非保留
CONSTRAINT	保留	保留	保留
CONSTRAINTS	非保留	保留	保留
CONSTRAINT_CATALOG	-	非保留	非保留
CONSTRAINT_NAME	-	非保留	非保留
CONSTRAINT_SCHEMA	-	非保留	非保留
CONSTRUCTOR	-	保留	-
CONTAINS	-	非保留	-
CONTENT	非保留	-	-
CONTINUE	非保留	保留	保留
CONVERSION	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
CONVERT	-	非保留	保留
COORDINATOR	非保留	-	-
COPY	非保留	-	-
CORRESPONDING	-	保留	保留
COST	非保留	-	-
COUNT	-	非保留	保留
CREATE	保留	保留	保留
CROSS	保留(可以是函数或类型)	保留	保留
CSV	非保留	-	-
CUBE	-	保留	-
CURRENT	非保留	保留	保留
CURRENT_CATALOG	保留	-	-
CURRENT_DATE	保留	保留	保留
CURRENT_PATH	-	保留	-
CURRENT_ROLE	保留	保留	-
CURRENT_SCHEMA	保留(可以是函数或类型)	-	-
CURRENT_TIME	保留	保留	保留
CURRENT_TIMESTAMP	保留	保留	保留
CURRENT_USER	保留	保留	保留
CURSOR	非保留	保留	保留
CURSOR_NAME	-	非保留	非保留
CYCLE	非保留	保留	-
DATA	非保留	保留	非保留
DATE_FORMAT	非保留	-	-
DATABASE	非保留	-	-
DATAFILE	非保留	-	-
DATE	非保留(不能是函数或类型)	保留	保留

关键字	GaussDB(DWS)	SQL:1999	SQL-92
DATETIME_INTERVAL_CODE	-	非保留	非保留
DATETIME_INTERVAL_PRECISION	-	非保留	非保留
DAY	非保留	保留	保留
DBCOMPATIBILITY	非保留	-	-
DEALLOCATE	非保留	保留	保留
DEC	非保留(不能是函数或类型)	保留	保留
DECIMAL	非保留(不能是函数或类型)	保留	保留
DECLARE	非保留	保留	保留
DECODE	非保留(不能是函数或类型)	-	-
DEFAULT	保留	保留	保留
DEFAULTS	非保留	-	-
DEFERRABLE	保留	保留	保留
DEFERRED	非保留	保留	保留
DEFINED	-	非保留	-
DEFINER	非保留	非保留	-
DELETE	非保留	保留	保留
DELIMITER	非保留	-	-
DELIMITERS	非保留	-	-
DELTA	非保留	-	-
DEPTH	-	保留	-
DEREF	-	保留	-
DESC	保留	保留	保留
DESCRIBE	-	保留	保留
DESCRIPTOR	-	保留	保留
DESTROY	-	保留	-
DESTRUCTOR	-	保留	-
DETERMINISTIC	非保留	保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
DIAGNOSTICS	-	保留	保留
DICTIONARY	非保留	保留	-
DIRECT	非保留	-	-
DIRECTORY	非保留	-	-
DISABLE	非保留	-	-
DISCARD	非保留	-	-
DISCONNECT	-	保留	保留
DISPATCH	-	非保留	-
DISTINCT	保留	保留	保留
DISTRIBUTE	非保留	-	-
DISTRIBUTION	非保留	-	-
DO	保留	-	-
DOCUMENT	非保留	-	-
DOMAIN	非保留	保留	保留
DOUBLE	非保留	保留	保留
DROP	非保留	保留	保留
DYNAMIC	-	保留	-
DYNAMIC_FUNCTION	-	非保留	非保留
DYNAMIC_FUNCTION_CODE	-	非保留	-
EACH	非保留	保留	-
ELASTIC	非保留	-	-
ELSE	保留	保留	保留
ENABLE	非保留	-	-
ENCODING	非保留	-	-
ENCRYPTED	非保留	-	-
END	保留	保留	保留
END-EXEC	-	保留	保留
ENFORCED	非保留	-	-
ENUM	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
EOL	非保留	-	-
EQUALS	-	保留	-
ERRORS	非保留	-	-
ESCAPE	非保留	保留	保留
ESCAPING	非保留	-	-
EVERY	非保留	保留	-
EXCEPT	保留	保留	保留
EXCEPTION	-	保留	保留
EXCHANGE	非保留	-	-
EXCLUDE	非保留	-	-
EXCLUDING	非保留	-	-
EXCLUSIVE	非保留	-	-
EXEC	-	保留	保留
EXECUTE	非保留	保留	保留
EXISTING	-	非保留	-
EXISTS	非保留(不能是函数或类型)	非保留	保留
EXPIRATION	非保留	-	-
EXPLAIN	非保留	-	-
EXTENSION	非保留	-	-
EXTERNAL	非保留	保留	保留
EXTRACT	非保留(不能是函数或类型)	非保留	保留
FALSE	保留	保留	保留
FAMILY	非保留	-	-
FAST	非保留	-	-
FENCED	非保留	-	-
FETCH	保留	保留	保留
FILEHEADER	非保留	-	-
FILL_MISSING_FIELDS	非保留	-	-
FINAL	-	非保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
FIRST	非保留	保留	保留
FIXED	非保留	保留	保留
FLOAT	非保留(不能是函数或类型)	保留	保留
FOLLOWING	非保留	-	-
FOR	保留	保留	保留
FORCE	非保留	-	-
FOREIGN	保留	保留	保留
FORMATTER	非保留	-	-
FORTRAN	-	非保留	非保留
FORWARD	非保留	-	-
FOUND	-	保留	保留
FREE	-	保留	-
FREEZE	保留(可以是函数或类型)	-	-
FROM	保留	保留	保留
FULL	保留(可以是函数或类型)	保留	保留
FUNCTION	非保留	保留	-
FUNCTIONS	非保留	-	-
G	-	非保留	-
GENERAL	-	保留	-
GENERATED	-	非保留	-
GET	-	保留	保留
GLOBAL	非保留	保留	保留
GO	-	保留	保留
GOTO	-	保留	保留
GRANT	保留	保留	保留
GRANTED	非保留	非保留	-
GREATEST	非保留(不能是函数或类型)	-	-
GROUP	保留	保留	保留

关键字	GaussDB(DWS)	SQL:1999	SQL-92
GROUPING	-	保留	-
HANDLER	非保留	-	-
HAVING	保留	保留	保留
HEADER	非保留	-	-
HIERARCHY	-	非保留	-
HOLD	非保留	非保留	-
HOST	-	保留	-
HOUR	非保留	保留	保留
IDENTIFIED	非保留	-	-
IDENTITY	非保留	保留	保留
IF	非保留(不能是函数或类型)	-	-
IFNULL	非保留(不能是函数或类型)	-	-
IGNORE	-	保留	-
IGNORE_EXTRA_DATA	非保留	-	-
ILIKE	保留(可以是函数或类型)	-	-
IMMEDIATE	非保留	保留	保留
IMMUTABLE	非保留	-	-
IMPLEMENTATION	-	非保留	-
IMPLICIT	非保留	-	-
IN	保留	保留	保留
INCLUDING	非保留	-	-
INCREMENT	非保留	-	-
INDEX	非保留	-	-
INDEXES	非保留	-	-
INDICATOR	-	保留	保留
INFIX	-	非保留	-
INHERIT	非保留	-	-
INHERITS	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
INITIAL	非保留	-	-
INITIALIZE	-	保留	-
INITIALLY	保留	保留	保留
INITRANS	非保留	-	-
INLINE	非保留	-	-
INNER	保留(可以是函数或类型)	保留	保留
INOUT	非保留(不能是函数或类型)	保留	-
INPUT	非保留	保留	保留
INSENSITIVE	非保留	非保留	保留
INSERT	非保留	保留	保留
INSTANCE	-	非保留	-
INSTANTIABLE	-	非保留	-
INSTEAD	非保留	-	-
INT	非保留(不能是函数或类型)	保留	保留
INTEGER	非保留(不能是函数或类型)	保留	保留
INTERNAL	保留	-	-
INTERSECT	保留	保留	保留
INTERVAL	非保留(不能是函数或类型)	保留	保留
INTO	保留	保留	保留
INVOKER	非保留	非保留	-
IS	保留	保留	保留
ISNULL	非保留(不能是函数或类型)	-	-
ISOLATION	非保留	保留	保留
ITERATE	-	保留	-
JOIN	保留(可以是函数或类型)	保留	保留
K	-	非保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
KEY	非保留	保留	保留
KEY_MEMBER	-	非保留	-
KEY_TYPE	-	非保留	-
LABEL	非保留	-	-
LANGUAGE	非保留	保留	保留
LARGE	非保留	保留	-
LAST	非保留	保留	保留
LATERAL	-	保留	-
LC_COLLATE	非保留	-	-
LC_CTYPE	非保留	-	-
LEADING	保留	保留	保留
LEAKPROOF	非保留	-	-
LEAST	非保留(不能是函数或类型)	-	-
LEFT	保留(可以是函数或类型)	保留	保留
LENGTH	-	非保留	非保留
LESS	保留	保留	-
LEVEL	非保留	保留	保留
LIKE	保留(可以是函数或类型)	保留	保留
LIMIT	保留	保留	-
LISTEN	非保留	-	-
LOAD	非保留	-	-
LOCAL	非保留	保留	保留
LOCALTIME	保留	保留	-
LOCALTIMESTAMP	保留	保留	-
LOCATION	非保留	-	-
LOCATOR	-	保留	-
LOCK	非保留	-	-
LOG	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
LOGGING	非保留	-	-
LOGIN	非保留	-	-
LOOP	非保留	-	-
LOWER	-	非保留	保留
M	-	非保留	-
MAP	-	保留	-
MAPPING	非保留	-	-
MATCH	非保留	保留	保留
MATCHED	非保留	-	-
MATERIALIZED	非保留	-	-
MAX	-	非保留	保留
MAXEXTENTS	非保留	-	-
MAXSIZE	非保留	-	-
MAXTRANS	非保留	-	-
MAXVALUE	保留	-	-
MERGE	非保留	-	-
MESSAGE_LENGTH	-	非保留	非保留
MESSAGE_OCTET_LENGTH	-	非保留	非保留
MESSAGE_TEXT	-	非保留	非保留
METHOD	-	非保留	-
MIN	-	非保留	保留
MINEXTENTS	非保留	-	-
MINUS	保留	-	-
MINUTE	非保留	保留	保留
MINVALUE	非保留	-	-
MOD	-	非保留	-
MODE	非保留	-	-
MODIFIES	-	保留	-
MODIFY	保留	保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
MODULE	-	保留	保留
MONTH	非保留	保留	保留
MORE	-	非保留	非保留
MOVE	非保留	-	-
MOVEMENT	非保留	-	-
MUMPS	-	非保留	非保留
NAME	非保留	非保留	非保留
NAMES	非保留	保留	保留
NATIONAL	非保留(不能是函数或类型)	保留	保留
NATURAL	保留(可以是函数或类型)	保留	保留
NCHAR	非保留(不能是函数或类型)	保留	保留
NCLOB	-	保留	-
NEW	-	保留	-
NEXT	非保留	保留	保留
NLSSORT	保留	-	-
NO	非保留	保留	保留
NOCOMPRESS	非保留	-	-
NOCYCLE	非保留	-	-
NODE	非保留	-	-
NOLOGGING	非保留	-	-
NOLOGIN	非保留	-	-
NOMAXVALUE	非保留	-	-
NOMINVALUE	非保留	-	-
NONE	非保留(不能是函数或类型)	保留	-
NOT	保留	保留	保留
NOTHING	非保留	-	-
NOTIFY	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
NOTNULL	保留(可以是函数或类型)	-	-
NOWAIT	非保留	-	-
NULL	保留	保留	保留
NULLABLE	-	非保留	非保留
NULLIF	非保留(不能是函数或类型)	非保留	保留
NULLS	非保留	-	-
NUMBER	非保留(不能是函数或类型)	非保留	非保留
NUMERIC	非保留(不能是函数或类型)	保留	保留
NUMSTR	非保留	-	-
NVARCHAR2	非保留(不能是函数或类型)	-	-
NVL	非保留(不能是函数或类型)	-	-
OBJECT	非保留	保留	-
OCTET_LENGTH	-	非保留	保留
OF	非保留	保留	保留
OFF	非保留	保留	-
OFFSET	保留	-	-
OIDS	非保留	-	-
OLD	-	保留	-
ON	保留	保留	保留
ONLY	保留	保留	保留
OPEN	-	保留	保留
OPERATION	-	保留	-
OPERATOR	非保留	-	-
OPTIMIZATION	非保留	-	-
OPTION	非保留	保留	保留
OPTIONS	非保留	非保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
OR	保留	保留	保留
ORDER	保留	保留	保留
ORDINALITY	-	保留	-
OUT	非保留(不能是函数或类型)	保留	-
OUTER	保留(可以是函数或类型)	保留	保留
OUTPUT	-	保留	保留
OVER	非保留	-	-
OVERLAPS	保留(可以是函数或类型)	非保留	保留
OVERLAY	非保留(不能是函数或类型)	非保留	-
OVERRIDING	-	非保留	-
OWNED	非保留	-	-
OWNER	非保留	-	-
PACKAGE	非保留	-	-
PAD	-	保留	保留
PARAMETER	-	保留	-
PARAMETERS	-	保留	-
PARAMETER_MODE	-	非保留	-
PARAMETER_NAME	-	非保留	-
PARAMETER_ORDINAL_POSITION	-	非保留	-
PARAMETER_SPECIFIC_CATALOG	-	非保留	-
PARAMETER_SPECIFIC_NAME	-	非保留	-
PARAMETER_SPECIFIC_SCHEMA	-	非保留	-
PARSER	非保留	-	-
PARTIAL	非保留	保留	保留
PARTITION	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
PARTITIONS	非保留	-	-
PASCAL	-	非保留	非保留
PASSING	非保留	-	-
PASSWORD	非保留	-	-
PATH	-	保留	-
PCTFREE	非保留	-	-
PER	非保留	-	-
PERM	非保留	-	-
PERCENT	非保留	-	-
PERFORMANCE	保留	-	-
PLACING	保留	-	-
PLAN	保留	-	-
PLANS	非保留	-	-
PLI	-	非保留	非保留
POLICY	非保留	-	-
POOL	非保留	-	-
POSITION	非保留(不能是函数或类型)	非保留	保留
POSTFIX	-	保留	-
PRECEDING	非保留	-	-
PRECISION	非保留(不能是函数或类型)	保留	保留
PREFERRED	非保留	-	-
PREFIX	非保留	保留	-
PREORDER	-	保留	-
PREPARE	非保留	保留	保留
PREPARED	非保留	-	-
PRESERVE	非保留	保留	保留
PRIMARY	保留	保留	保留
PRIOR	非保留	保留	保留
PRIVATE	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
PRIVILEGE	非保留	-	-
PRIVILEGES	非保留	保留	保留
PROCEDURAL	非保留	-	-
PROCEDURE	保留	保留	保留
PROFILE	非保留	-	-
PUBLIC	-	保留	保留
QUERY	非保留	-	-
QUOTE	非保留	-	-
RANGE	非保留	-	-
RAW	非保留	-	-
READ	非保留	保留	保留
READS	-	保留	-
REAL	非保留(不能是函数或类型)	保留	保留
REASSIGN	非保留	-	-
REBUILD	非保留	-	-
RECHECK	非保留	-	-
RECURSIVE	非保留	保留	-
REF	非保留	保留	-
REFRESH	非保留	-	-
REFERENCES	保留	保留	保留
REFERENCING	-	保留	-
REINDEX	非保留	-	-
REJECT	保留	-	-
RELATIVE	非保留	保留	保留
RELEASE	非保留	-	-
RELOPTIONS	非保留	-	-
REMOTE	非保留	-	-
RENAME	非保留	-	-
REPEATABLE	非保留	非保留	非保留

关键字	GaussDB(DWS)	SQL:1999	SQL-92
REPLACE	非保留	-	-
REPLICA	非保留	-	-
RESET	非保留	-	-
RESIZE	非保留	-	-
RESOURCE	非保留	-	-
RESTART	非保留	-	-
RESTRICT	非保留	保留	保留
RESULT	-	保留	-
RETURN	非保留	保留	-
RETURNED_LENGTH	-	非保留	非保留
RETURNED_OCTET_LENGTH	-	非保留	非保留
RETURNED_SQLSTATE	-	非保留	非保留
RETURNING	保留	-	-
RETURNS	非保留	保留	-
REUSE	非保留	-	-
REVOKE	非保留	保留	保留
RIGHT	保留(可以是函数或类型)	保留	保留
ROLE	非保留	保留	-
ROLLBACK	非保留	保留	保留
ROLLUP	-	保留	-
ROUTINE	-	保留	-
ROUTINE_CATALOG	-	非保留	-
ROUTINE_NAME	-	非保留	-
ROUTINE_SCHEMA	-	非保留	-
ROW	非保留(不能是函数或类型)	保留	-
ROWS	非保留	保留	保留
ROW_COUNT	-	非保留	非保留
RULE	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
SAVEPOINT	非保留	保留	-
SCALE	-	非保留	非保留
SCHEMA	非保留	保留	保留
SCHEMA_NAME	-	非保留	非保留
SCOPE	-	保留	-
SCROLL	非保留	保留	保留
SEARCH	非保留	保留	-
SECOND	非保留	保留	保留
SECTION	-	保留	保留
SECURITY	非保留	非保留	-
SELECT	保留	保留	保留
SELF	-	非保留	-
SENSITIVE	-	非保留	-
SEPARATOR	非保留	-	-
SEQUENCE	非保留	保留	-
SEQUENCES	非保留	-	-
SERIALIZABLE	非保留	非保留	非保留
SERVER	非保留	-	-
SERVER_NAME	-	非保留	非保留
SESSION	非保留	保留	保留
SESSION_USER	保留	保留	保留
SET	非保留	保留	保留
SETOF	非保留(不能是函数或类型)	-	-
SETS	-	保留	-
SHARE	非保留	-	-
SHIPPABLE	非保留	-	-
SHOW	非保留	-	-
SIMILAR	保留(可以是函数或类型)	非保留	-
SIMPLE	非保留	非保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
SIZE	非保留	保留	保留
SMALLDATETIME	非保留(不能是函数或类型)	-	-
SMALLDATETIME_FORMAT	非保留	-	-
SMALLINT	非保留(不能是函数或类型)	保留	保留
SNAPSHOT	非保留	-	-
SOME	保留	保留	保留
SOURCE	非保留	非保留	-
SPACE	-	保留	保留
SPECIFIC	-	保留	-
SPECIFICTYPE	-	保留	-
SPECIFIC_NAME	-	非保留	-
SPILL	非保留	-	-
SPLIT	非保留	-	-
SQL	-	保留	保留
SQLCODE	-	-	保留
SQLERROR	-	-	保留
SQLEXCEPTION	-	保留	-
SQLSTATE	-	保留	保留
SQLWARNING	-	保留	-
STABLE	非保留	-	-
STANDALONE	非保留	-	-
START	非保留	保留	-
STATE	-	保留	-
STATEMENT	非保留	保留	-
STATEMENT_ID	非保留	-	-
STATIC	-	保留	-
STATISTICS	非保留	-	-
STDIN	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
STDOUT	非保留	-	-
STORAGE	非保留	-	-
STORE	非保留	-	-
STRICT	非保留	-	-
STRIP	非保留	-	-
STRUCTURE	-	保留	-
STYLE	-	非保留	-
SUBCLASS_ORIGIN	-	非保留	非保留
SUBLIST	-	非保留	-
SUBSTRING	非保留(不能是函数或类型)	非保留	保留
SUM	-	非保留	保留
SUPERUSER	非保留	-	-
SYMMETRIC	保留	非保留	-
SYNONYM	非保留	-	-
SYS_REFCURSOR	非保留	-	-
SYSDATE	保留	-	-
SYSID	非保留	-	-
SYSTEM	非保留	非保留	-
SYSTEM_USER	-	保留	保留
TABLE	保留	保留	保留
TABLES	非保留	-	-
TABLE_NAME	-	非保留	非保留
TEMP	非保留	-	-
TEMPLATE	非保留	-	-
TEMPORARY	非保留	保留	保留
TERMINATE	-	保留	-
TEXT	非保留	-	-
THAN	非保留	保留	-
THEN	保留	保留	保留

关键字	GaussDB(DWS)	SQL:1999	SQL-92
TIME	非保留(不能是函数或类型)	保留	保留
TIME_FORMAT	非保留	-	-
TIMESTAMP	非保留(不能是函数或类型)	保留	保留
TIMESTAMPADD	非保留(不能是函数或类型)	-	-
TIMESTAMPDIFF	非保留(不能是函数或类型)	-	-
TIMESTAMP_FORMAT	非保留	-	-
TIMEZONE_HOUR	-	保留	保留
TIMEZONE_MINUTE	-	保留	保留
TINYINT	非保留(不能是函数或类型)	-	-
TO	保留	保留	保留
TRAILING	保留	保留	保留
TRANSACTION	非保留	保留	保留
TRANSACTIONS_COMMITTED	-	非保留	-
TRANSACTIONS_ROLLED_BACK	-	非保留	-
TRANSACTION_ACTIVE	-	非保留	-
TRANSFORM	-	非保留	-
TRANSFORMS	-	非保留	-
TRANSLATE	-	非保留	保留
TRANSLATION	-	保留	保留
TREAT	非保留(不能是函数或类型)	保留	-
TRIGGER	非保留	保留	-
TRIGGER_CATALOG	-	非保留	-
TRIGGER_NAME	-	非保留	-
TRIGGER_SCHEMA	-	非保留	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
TRIM	非保留(不能是函数或类型)	非保留	保留
TRUE	保留	保留	保留
TRUNCATE	非保留	-	-
TRUSTED	非保留	-	-
TSTAG	保留，该字段仅在IoT数仓中使用	-	-
TSTIME	保留，该字段仅在IoT数仓中使用	-	-
TSFIELD	保留，该字段仅在IoT数仓中使用	-	-
TYPE	非保留	非保留	非保留
TYPES	非保留	-	-
UESCAPE	-	-	-
UNBOUNDED	非保留	-	-
UNCOMMITTED	非保留	非保留	非保留
UNDER	-	保留	-
UNENCRYPTED	非保留	-	-
UNION	保留	保留	保留
UNIQUE	保留	保留	保留
UNKNOWN	非保留	保留	保留
UNLIMITED	非保留	-	-
UNLISTEN	非保留	-	-
UNLOCK	非保留	-	-
UNLOGGED	非保留	-	-
UNNAMED	-	非保留	非保留
UNNEST	-	保留	-
UNTIL	非保留	-	-
UNUSABLE	非保留	-	-
UPDATE	非保留	保留	保留
UPPER	-	非保留	保留
USAGE	-	保留	保留

关键字	GaussDB(DWS)	SQL:1999	SQL-92
USER	保留	保留	保留
USER_DEFINED_TYPE_CATALOG	-	非保留	-
USER_DEFINED_TYPE_NAME	-	非保留	-
USER_DEFINED_TYPE_SCHEMA	-	非保留	-
USING	保留	保留	保留
VACUUM	非保留	-	-
VALID	非保留	-	-
VALIDATE	非保留	-	-
VALIDATION	非保留	-	-
VALIDATOR	非保留	-	-
VALUE	非保留	保留	保留
VALUES	非保留(不能是函数或类型)	保留	保留
VARCHAR	非保留(不能是函数或类型)	保留	保留
VARCHAR2	非保留(不能是函数或类型)	-	-
VARIABLE	-	保留	-
VARIADIC	保留	-	-
VARYING	非保留	保留	保留
VCGROUP	非保留	-	-
VERBOSE	保留(可以是函数或类型)	-	-
VERIFY	非保留	-	-
VERSION	非保留	-	-
VIEW	非保留	保留	保留
VOLATILE	非保留	-	-
WHEN	保留	保留	保留
WHENEVER	-	保留	保留
WHERE	保留	保留	保留

关键字	GaussDB(DWS)	SQL:1999	SQL-92
WHITESPACE	非保留	-	-
WINDOW	保留	-	-
WITH	保留	保留	保留
WITHIN	非保留	-	-
WITHOUT	非保留	保留	-
WORK	非保留	保留	保留
WORKLOAD	非保留	-	-
WRAPPER	非保留	-	-
WRITE	非保留	保留	保留
XML	非保留	-	-
XMLATTRIBUTES	非保留(不能是函数或类型)	-	-
XMLCONCAT	非保留(不能是函数或类型)	-	-
XMLELEMENT	非保留(不能是函数或类型)	-	-
XMLEXISTS	非保留(不能是函数或类型)	-	-
XMLFOREST	非保留(不能是函数或类型)	-	-
XMLNAMESPACES	非保留(不能是函数或类型)	-	-
XMLPARSE	非保留(不能是函数或类型)	-	-
XMLPI	非保留(不能是函数或类型)	-	-
XMLROOT	非保留(不能是函数或类型)	-	-
XMLSERIALIZE	非保留(不能是函数或类型)	-	-
XMLTABLE	非保留(不能是函数或类型)	-	-
YEAR	非保留	保留	保留
YES	非保留	-	-

关键字	GaussDB(DWS)	SQL:1999	SQL-92
ZONE	非保留	保留	保留

4 数据类型

4.1 数值类型

数值类型也叫数字类型。由1、2、4或8字节的整数以及4或8字节的浮点数和可选精度小数组成。

对应的数字操作符和相关函数，请参见**数字操作函数和操作符**。

GaussDB(DWS)支持的数值类型按精度可以分为：整数类型，任意精度型，浮点类型和序列整型。

整数类型

TINYINT、SMALLINT、INTEGER、BINARY_INTEGER和BIGINT类型存储整个数值（不带有小数部分），也就是整数。如果尝试存储超出范围以外的数值将会导致错误。

常用的类型是INTEGER，一般只有取值范围确定不超过SMALLINT的情况下，才会使用SMALLINT类型。而只有在INTEGER的范围不够的时候才使用BIGINT，因为前者相对快得多。

表 4-1 整数类型

名称	描述	存储空间	范围
TINYINT	微整数，别名为INT1。	1字节	0 ~ 255
SMALLINT	小范围整数，别名为INT2。	2字节	-32,768 ~ +32,767
INTEGER	常用的整数，别名为INT4。	4字节	-2,147,483,648 ~ +2,147,483,647
BINARY_INTEGER	常用的整数INTEGER的别名，为兼容Oracle类型。	4字节	-2,147,483,648 ~ +2,147,483,647

名称	描述	存储空间	范围
BIGINT	大范围的整数，别名为INT8。	8字节	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

示例：

创建带有TINYINT、INTEGER、BIGINT类型数据的表。

```
CREATE TABLE int_type_t1
(
    a TINYINT,
    b TINYINT,
    c INTEGER,
    d BIGINT
);
```

插入数据。

```
INSERT INTO int_type_t1 VALUES(100, 10, 1000, 10000);
```

查看数据。

```
SELECT * FROM int_type_t1;
 a | b |  c   |   d
---+---+------+-------
 100 | 10 | 1000 | 10000
(1 row)
```

任意精度型

NUMBER类型能够用于存储对于精度位数没有限制的数字，并且可以用于执行精确计算。当要求高精确度时，推荐使用这种类型来存储货币总量和其他类型的数量值。与整数类型相比，任意精度类型需要更大的存储空间，其存储效率、运算效率以及压缩比效果都要差一些。

NUMBER类型数值的范围是小数点右边部分的小数位数。NUMBER类型数值的精度是指整个数值包含的所有数字，也就是小数点左右两边的所有数字。所以，可以说数值23.1234的精度为6，范围是4。可以认为整数的范围是0。

使用Numeric/Decimal进行列定义时，建议指定该列的精度p（总位数）以及范围s（小数位数）。

如果数值的精度或者范围大于列的数据类型所声明的精度和范围，那么系统将会试图对这个值进行四舍五入。如果不能对数值进行四舍五入的处理来满足数据类型的限制，则会报错。

表 4-2 任意精度型

名称	描述	存储空间	范围
NUMERIC[(p[,s])], DECIMAL[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大131,072位，小数点后最大16,383位。

名称	描述	存储空间	范围
NUMBER[(p[,s])]	NUMERIC类型的别名，为兼容Oracle数据类型。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大131,072位，小数点后最大16,383位。

示例：

创建带有DECIMAL数值类型的表。

```
CREATE TABLE decimal_type_t1 (DT_COL1 DECIMAL(10,4));
```

插入数据。

```
INSERT INTO decimal_type_t1 VALUES(123456.122331);
INSERT INTO decimal_type_t1 VALUES(123456.452399);
```

查看数据。

```
SELECT * FROM decimal_type_t1;
   dt_col1
-------------
 123456.1223
 123456.4524
(2 rows)
```

浮点类型

浮点类型属于非精确，可变精度的数值类型。实际上，这些类型通常是对于二进制浮点算术（分别是单精度和双精度）的IEEE标准754的具体实现，在一定范围内由特定的处理器，操作系统和编译器所支持。

表 4-3 浮点类型

名称	描述	存储空间	范围
REAL, FLOAT4	单精度浮点数，不精准。	4字节	6位十进制数字精度。
DOUBLE PRECISION, FLOAT8	双精度浮点数，不精准。	8字节	1E-307~1E+308， 15位十进制数字精度。
FLOAT[(p)]	浮点数，不精准。精度p取值范围为[1,53]。 **说明** p为精度，表示总位数。	4字节或8字节	根据精度p不同选择REAL或DOUBLE PRECISION作为内部表示。如不指定精度，内部用DOUBLE PRECISION表示。

名称	描述	存储空间	范围
BINARY_DOUBLE	是DOUBLE PRECISION的别名，为兼容Oracle类型。	8字节	1E-307~1E+308， 15位十进制数字精度。
DEC[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。 **说明** p为总位数，s为小数位位数。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大131,072位，小数点后最大16,383位。
INTEGER[(p[,s])]	精度p取值范围为[1,1000]，标度s取值范围为[0,p]。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大131,072位，小数点后最大16,383位。

示例：

创建带有浮点类型的表。

```
CREATE TABLE float_type_t2
(
    FT_COL1 INTEGER,
    FT_COL2 FLOAT4,
    FT_COL3 FLOAT8,
    FT_COL4 FLOAT(3),
    FT_COL5 BINARY_DOUBLE,
    FT_COL6 DECIMAL(10,4),
    FT_COL7 INTEGER(6,3)
) DISTRIBUTE BY HASH ( ft_col1);
```

插入数据。

```
INSERT INTO float_type_t2 VALUES(10,10.365456,123456.1234,10.3214, 321.321, 123.123654, 123.123654);
```

查看数据。

```
SELECT * FROM float_type_t2;
 ft_col1 | ft_col2 |   ft_col3   | ft_col4 | ft_col5 | ft_col6  | ft_col7
---------+---------+-------------+---------+---------+----------+---------
      10 | 10.3655 | 123456.1234 | 10.3214 | 321.321 | 123.1237 | 123.124
(1 row)
```

序列整型

SMALLSERIAL，SERIAL和BIGSERIAL类型不是真正的类型，只是为在表中设置唯一标识而存在的概念。因此，创建一个整数字段，并且把它的缺省数值安排为从一个序列发生器读取。应用了一个NOT NULL约束以确保NULL不会被插入。在大多数情况下用户可能还希望附加一个UNIQUE或PRIMARY KEY约束避免意外地插入重复的数值。最

后，将序列发生器从属于那个字段，这样当该字段或表被删除的时候也一并删除该序列。目前只支持在创建表时指定SERIAL列，不可以在已有的表中增加SERIAL列。另外临时表也不支持创建SERIAL列。因为SERIAL不是真正的类型，也不可以将表中存在的列类型转化为SERIAL。

表 4-4 序列整型

名称	描述	存储空间	范围
SMALLSERIAL	二字节序列整型。	2字节	1 ~ 32,767
SERIAL	四字节序列整型。	4字节	1 ~ 2,147,483,647
BIGSERIAL	八字节序列整型。	8字节	1 ~ 9,223,372,036,854,775,807

示例：

创建带有序列类型的表。

```
CREATE TABLE smallserial_type_tab(a SMALLSERIAL);
```

插入数据。

```
INSERT INTO smallserial_type_tab VALUES(default);
```

再次插入数据。

```
INSERT INTO smallserial_type_tab VALUES(default);
```

查看数据。

```
SELECT * FROM smallserial_type_tab;
 a
---
 1
 2
(2 rows)
```

插入NULL值会报错。

```
INSERT INTO smallserial_type_tab VALUES(NULL);
ERROR:  dn_6001_6002: null value in column "a" violates not-null constraint
```

4.2 货币类型

货币类型存储带有固定小数精度的货币金额。表4-5中显示的范围假设有两位小数。可以以任意格式输入，包括整型、浮点型或者典型的货币格式（如“$1,000.00”）。根据区域字符集，输出一般是最后一种形式。

表 4-5 货币类型

名字	存储容量	描述	范围
money	8 字节	货币金额	-92233720368547758.08 到 +92233720368547758.07

numeric、int和bigint类型的值可以转化为money类型。如果从real和double precision类型转换到money类型，可以先转化为numeric类型，再转化为money类型，例如：

```
SELECT '12.34'::float8::numeric::money;
```

这种用法是不推荐使用的。浮点数不应该用来处理货币类型，因为小数点的位数可能会导致错误。

money类型的值可以转换为numeric类型而不丢失精度。转换为其他类型可能丢失精度，并且必须通过以下两步来完成：

```
SELECT '52093.89'::money::numeric::float8;
```

当一个money类型的值除以另一个money类型的值时，结果是double precision（也就是，一个纯数字，而不是money类型）；在运算过程中货币单位相互抵消。

4.3 布尔类型

表 4-6 布尔类型

名称	描述	存储空间	取值
BOOLEAN	布尔类型	1字节。	• true：真 • false：假 • null：未知（unknown）

“真”值的有效文本值是：

TRUE、't'、'true'、'y'、'yes'、'1'。

“假”值的有效文本值是：

FALSE、'f'、'false'、'n'、'no'、'0'。

使用TRUE和FALSE是比较规范的用法（也是SQL兼容的用法）。

示例

显示用字母t和f输出boolean值。

```
--创建表。
CREATE TABLE bool_type_t1
(
    BT_COL1 BOOLEAN,
    BT_COL2 TEXT
) DISTRIBUTE BY HASH(BT_COL2);

--插入数据。
INSERT INTO bool_type_t1 VALUES (TRUE, 'sic est');

INSERT INTO bool_type_t1 VALUES (FALSE, 'non est');

--查看数据。
SELECT * FROM bool_type_t1;
 bt_col1 | bt_col2
---------+---------
 t       | sic est
 f       | non est
```

```
(2 rows)

SELECT * FROM bool_type_t1 WHERE bt_col1 = 't';
 bt_col1 | bt_col2
---------+---------
 t       | sic est
(1 row)

--删除表。
DROP TABLE bool_type_t1;
```

4.4 字符类型

GaussDB(DWS)支持的字符类型请参见表4-7。字符串操作符和相关的内置函数请参见字符处理函数和操作符。

表 4-7 字符类型

名称	描述	长度	存储空间
CHAR(n) CHARACTER(n) NCHAR(n)	定长字符串，不足填充空格。	n是指字节长度，如不带精度n，默认精度为1。n小于10485761。	最大为10MB。
VARCHAR(n) CHARACTER VARYING(n)	变长字符串。	n是指字节长度，n小于10485761。	最大为10MB。
VARCHAR2(n)	变长字符串。是VARCHAR(n)类型的别名，为兼容Oracle类型。	n是指字节长度，n小于10485761。	最大为10MB。
NVARCHAR2(n)	变长字符串。	n是指字符长度，n小于10485761。	最大为10MB。
CLOB	变长字符串。文本大对象。是TEXT类型的别名，为兼容Oracle类型。	-	最大为1GB-8203B（即1073733621B）。
TEXT	变长字符串。	-	最大为1GB-8203B（即1073733621B）。

说明

- 除了每列的大小限制以外，每个元组的总大小也不可超过1GB-8023B（即1073733621B）。
- 对于字符串数据，建议使用变长字符串数据类型，并指定最大长度。请务必确保指定的最大长度大于需要存储的最大字符数，避免超出最大长度时出现字符截断现象。除非明确知道数据类型为固定长度字符串，否则，不建议使用CHAR(n)、NCHAR(n)、CHARACTER(n)等定长字符类型。在GaussDB(DWS)中，定长字符类型的运算会存在额外的存储和内存开销。

在GaussDB(DWS)里另外还有两种定长字符类型。在表4-8里显示。

其中，name类型只用在内部系统表中，作为存储标识符，该类型长度当前定为64字节（63可用字符加结束符）。不建议普通用户使用这种数据类型。name类型与其他数据类型进行对齐时（比如case when的多个分支中，其中一个分支返回name类型，其他类型返回text类型），可能会出现向name类型对齐，字符截断。如果不希望出现字符按照64位截断的情况，则需要将name类型强制类型转化为text类型。

类型"char"只用了一个字节的存储空间。它在系统内部主要用于系统表，主要作为简单化的枚举类型使用。

表 4-8 特殊字符类型

名称	描述	存储空间
name	用于对象名的内部类型。	64字节。
"char"	单字节内部类型。	1字节。

长度

如果把一个字段定义为char(n)或者varchar(n)， 代表该字段最大可容纳n个长度的数据。无论哪种类型，可设置的最大长度都不得超过10485760（即10MB）。

当数据长度超过指定的长度n时，会抛出错误"value too long"。也可通过指定数据类型，使超过长度的数据自动截断。

示例：

1. 创建表t1，指定其字段的字符类型。
```
CREATE TABLE t1 (a char(5),b varchar(5));
```
2. 向表t1插入数据时超过指定的字节长度报错。
```
INSERT INTO t1 VALUES('bookstore','123');
ERROR:  value too long for type character(5)
CONTEXT:  referenced column: a
```
3. 向表t1插入数据并明确超过指定字节长度后自动截断。
```
INSERT INTO t1 VALUES('bookstore'::char(5),'12345678'::varchar(5));
INSERT 0 1

SELECT a,b FROM t1;
   a   |   b
-------+-------
 books | 12345
(1 row)
```

定长与变长

所有字符类型根据长度是否固定可以分为定长字符串与变长字符串两大类。

- 对于定长字符串，长度必须确定，如果不指定长度，则默认长度1；如果数据长度不足，会在尾部自动填充空格，用以存储和显示；但这部分填充的数据是无意义的，实际使用中会被忽略，如比较、排序或类型转换。
- 对于变长字符串，若指定长度，则为最大可存储数据长度；如果不指定长度，则认为该字段支持任意长度。

示例：

1. 创建表t2，指定其字段的字符类型。
```
CREATE TABLE t2 (a char(5),b varchar(5));
```
2. 向表t2插入数据并查询字段a的字节长度。因建表时指定a的字符类型为char(5)且是定长字符串，长度不足，填充空格，所以查询的字节长度为5。
```
INSERT INTO t2 VALUES('abc','abc');
INSERT 0 1

SELECT a,lengthb(a),b FROM t2;
   a   | lengthb |  b
-------+---------+-----
 abc   |       5 | abc
(1 row)
```
3. 用函数转换后查询字段a的实际字节长度为3。
```
SELECT a = b from t2;
 ?column?
----------
 t
(1 row)

SELECT cast(a as text) as val,lengthb(val) FROM t2;
 val | lengthb
-----+---------
 abc |       3
(1 row)
```

字节数和字符数

VARCHAR2(n)和NVARCHAR2(n)中长度含义不同，需区别对待。

- VARCHAR2中n为字节长度。
- NVARCHAR2中n为字符长度。

  说明

  以UTF8编码的数据库为例，字母占1个字节，汉字占3个字节，VARCHAR2(6)可以存放6个字母或2个汉字，而NVARCHAR2(6)可以存放6个字母或6个汉字。

示例：

1. 创建表t3，指定其字段的字符类型。
```
CREATE TABLE t3 (a varchar2(6),b nvarchar2(6));
```
2. 向表t3插入数据，超出长度报错。（以UTF8字符编码的数据库中操作为例）
```
INSERT INTO t3 values('产品名','auto');
ERROR:  value too long for type character varying(6)
CONTEXT:  referenced column: a

INSERT INTO t3 values('auto','产品名abcd');
ERROR:  dn_6003_6004: value too long for type nvarchar2(6)
CONTEXT:  referenced column: b
```
3. 调整插入数据长度后向表t3插入数据成功。
```
INSERT INTO t3 values('产品','产品名abc');
INSERT 0 1

SELECT a,b from t3;
  a   |    b
------+----------
 产品 | 产品名abc
```

空串与 NULL

Oracle兼容模式下，不区分空串与NULL，执行语句查询或数据导入时会将空串处理为NULL。

由于空串默认被处理为NULL，那就不能使用 = '' 作为查询条件，也不能用 is ''。虽然不会有语法错误，但是不会有结果集返回。正确的用法是 is null，不等于就是 is not null 。

示例：

1. 创建表t4，指定其字段的字符类型。
```
CREATE TABLE t4 (a text);
```
2. 向表t4插入数据，插入值中包含空串和NULL。
```
INSERT INTO t4 VALUES('abc'),(''),(null);
INSERT 0 3
```
3. 查询表t4中是否存在空值。
```
SELECT a,a isnull FROM t4;
  a  | ?column?
-----+----------
     | t
     | t
 abc | f
(3 rows)

SELECT a,a isnull FROM t4 WHERE a is null;
 a | ?column?
---+----------
   | t
   | t
(2 rows)
```

TD与MySQL兼容模式下，区分空串与null。

- TD兼容模式下示例：
```
SELECT  '' is null , null is null;
 isnull |  isnull
--------+----------
 f      | t
(1 rows)
```
- MySQL兼容模式下示例：
```
SELECT  '' is null , null is null;
 isnull |  isnull
--------+----------
 f      | t
(1 rows)
```

4.5 二进制类型

GaussDB(DWS)支持的二进制类型请参见**表4-9**。

表 4-9 二进制类型

名称	描述	存储空间
BLOB	二进制大对象 目前BLOB支持的外部存取接口仅为： ● DBMS_LOB.GETLENGTH ● DBMS_LOB.READ ● DBMS_LOB.WRITE ● DBMS_LOB.WRITEAPPEND ● DBMS_LOB.COPY ● DBMS_LOB.ERASE 这些接口详细说明请参见DBMS_LOB。 **说明** 列存不支持BLOB类型	最大为1G-8023B（即1073733621B）。
RAW	变长的十六进制类型 **说明** 列存不支持RAW类型	4字节加上实际的十六进制字符串。最大为1G-8023B（即1073733621B）。
BYTEA	变长的二进制字符串	4字节加上实际的二进制字符串。最大为1G-8023B（即1073733621B）。

说明

除了每列的大小限制以外，每个元组的总大小也不可超过1G-8203字节。

示例

创建表：

```
CREATE TABLE blob_type_t1
(
    BT_COL1 INTEGER,
    BT_COL2 BLOB,
    BT_COL3 RAW,
    BT_COL4 BYTEA
) DISTRIBUTE BY REPLICATION;
```

插入数据：

```
INSERT INTO blob_type_t1 VALUES(10,empty_blob(),HEXTORAW('DEADBEEF'),E'\\xDEADBEEF');
```

查询表中的数据：

```
SELECT * FROM blob_type_t1;
 bt_col1 | bt_col2 | bt_col3  |  bt_col4
---------+---------+----------+------------
      10 |         | DEADBEEF | \xdeadbeef
(1 row)
```

删除表：

```
DROP TABLE blob_type_t1;
```

4.6 日期/时间类型

GaussDB(DWS)支持的日期/时间类型请参见**表4-10**。该类型的操作符和内置函数请参见**时间、日期处理函数和操作符**。

说明

如果其他的数据库时间格式和GaussDB(DWS)的时间格式不一致，可通过修改配置参数**DateStyle**的值来保持一致。

表 4-10 日期/时间类型

名称	描述	存储空间
DATE	Oracle兼容模式下等价于timestamp(0)，记录日期和时间。 Teradata和MySQL兼容模式下，记录日期。	Oracle兼容模式下，占存储空间8字节 其他模式下，占存储空间4字节
TIME [(p)] [WITHOUT TIME ZONE]	只用于一日内时间。 p表示小数点后的精度，取值范围为0~6。	8字节
TIME [(p)] [WITH TIME ZONE]	只用于一日内时间，带时区。 p表示小数点后的精度，取值范围为0~6。	12字节
TIMESTAMP[(p)] [WITHOUT TIME ZONE]	日期和时间。 p表示小数点后的精度，取值范围为0~6。	8字节
TIMESTAMP[(p)] [WITH TIME ZONE]	日期和时间，带时区。TIMESTAMP的别名为TIMESTAMPTZ。 p表示小数点后的精度，取值范围为0~6。	8字节
SMALLDATETIME	日期和时间，不带时区。 精确到分钟，秒位大于等于30秒进一位。	8字节
INTERVAL DAY (l) TO SECOND (p)	时间间隔，X天X小时X分X秒。 • l：天数的精度，取值范围为0~6。为适配Oracle语法，未实现具体功能。 • p：秒数的精度，取值范围为0~6。小数末尾的零不显示。	16字节

名称	描述	存储空间
INTERVAL [FIELDS] [(p)]	时间间隔。 ● fields：可以是YEAR，MONTH，DAY，HOUR，MINUTE，SECOND，DAY TO HOUR，DAY TO MINUTE，DAY TO SECOND，HOUR TO MINUTE，HOUR TO SECOND，MINUTE TO SECOND。 ● p：秒数的精度，取值范围为0~6，且fields为SECOND，DAY TO SECOND，HOUR TO SECOND或MINUTE TO SECOND时，参数p才有效。小数末尾的零不显示。	12字节
reltime	相对时间间隔。格式为： X years X mons X days XX:XX:XX。 ● 采用儒略历计时，规定一年为365.25天，一个月为30天，计算输入值对应的相对时间间隔，输出采用POSTGRES格式。	4字节

示例：

```
--创建表。
CREATE TABLE date_type_tab(coll date);

--插入数据。
INSERT INTO date_type_tab VALUES (date '12-10-2010');

--查看数据。
SELECT * FROM date_type_tab;
        coll
---------------------
 2010-12-10 00:00:00
(1 row)

--删除表。
DROP TABLE date_type_tab;

--创建表。
CREATE TABLE time_type_tab (da time without time zone ,dai time with time zone,dfgh timestamp without time zone,dfga timestamp with time zone, vbg smalldatetime);

--插入数据。
INSERT INTO time_type_tab VALUES ('21:21:21','21:21:21 pst','2010-12-12','2013-12-11 pst','2003-04-12 04:05:06');

--查看数据。
SELECT * FROM time_type_tab;
    da    |     dai     |        dfgh         |         dfga          |        vbg
----------+-------------+---------------------+-----------------------+---------------------
 21:21:21 | 21:21:21-08 | 2010-12-12 00:00:00 | 2013-12-11 16:00:00+08 | 2003-04-12 04:05:00
(1 row)

--删除表。
```

```
DROP TABLE time_type_tab;

--创建表。
CREATE TABLE day_type_tab (a int,b INTERVAL DAY(3) TO SECOND (4));

--插入数据。
INSERT INTO day_type_tab VALUES (1, INTERVAL '3' DAY);

--查看数据。
SELECT * FROM day_type_tab;
 a |   b
---+--------
 1 | 3 days
(1 row)

--删除表。
DROP TABLE day_type_tab;

--创建表。
CREATE TABLE year_type_tab(a int, b interval year (6));

--插入数据。
INSERT INTO year_type_tab VALUES(1,interval '2' year);

--查看数据。
SELECT * FROM year_type_tab;
 a |    b
---+---------
 1 | 2 years
(1 row)

--删除表。
DROP TABLE year_type_tab;
```

日期输入

日期和时间的输入几乎可以是任何合理的格式，包括ISO-8601格式、SQL-兼容格式、传统POSTGRES格式或者其它的形式。系统支持按照日、月、年的顺序自定义日期输入。如果把DateStyle参数设置为MDY就按照“月-日-年”解析，设置为DMY就按照“日-月-年”解析，设置为YMD就按照“年-月-日”解析。

日期的文本输入需要加单引号包围，语法如下：

type [(p)] 'value'

可选的精度声明中的p是一个整数，表示在秒域中小数部分的位数。**表4-11**显示了date类型的输入方式。

表 4-11 日期输入方式

例子	描述
1999-01-08	ISO 8601格式（建议格式），任何方式下都是1999年1月8号。
January 8, 1999	在任何datestyle输入模式下都无歧义。
1/8/1999	有歧义，在MDY模式下是一月八号，在DMY模式下是八月一号。
1/18/1999	MDY模式下是一月十八日，其它模式下被拒绝。

例子	描述
01/02/03	● MDY模式下的2003年1月2日。 ● DMY模式下的2003年2月1日。 ● YMD模式下的2001年2月3日。
1999-Jan-08	任何模式下都是1月8日。
Jan-08-1999	任何模式下都是1月8日。
08-Jan-1999	任何模式下都是1月8日。
99-Jan-08	YMD模式下是1月8日，否则错误。
08-Jan-99	一月八日，除了在YMD模式下是错误的之外。
Jan-08-99	一月八日，除了在YMD模式下是错误的之外。
19990108	ISO 8601；任何模式下都是1999年1月8日。
990108	ISO 8601；任何模式下都是1999年1月8日。
1999.008	年和年里的第几天。
J2451187	儒略日。
January 8, 99 BC	公元前99年。

示例：

```
--创建表。
CREATE TABLE date_type_tab(coll date);

--插入数据。
INSERT INTO date_type_tab VALUES (date '12-10-2010');

--查看数据。
SELECT * FROM date_type_tab;
        coll
---------------------
 2010-12-10 00:00:00
(1 row)

--查看日期格式。
SHOW datestyle;
 DateStyle
-----------
 ISO, MDY
(1 row)

--设置日期格式。
SET datestyle='YMD';
SET

--插入数据。
INSERT INTO date_type_tab VALUES(date '2010-12-11');

--查看数据。
SELECT * FROM date_type_tab;
        coll
---------------------
 2010-12-10 00:00:00
```

```
 2010-12-11 00:00:00
(2 rows)

--删除表。
DROP TABLE date_type_tab;
```

时间

时间类型包括time [(p)] without time zone和time [(p)] with time zone。如果只写time等效于time without time zone。

如果在time without time zone类型的输入中声明了时区，则会忽略这个时区。

时间输入类型的详细信息请参见**表4-12**，时区输入类型的详细信息请参见**表4-13**。

表 4-12 时间输入

例子	描述
05:06.8	ISO 8601
4:05:06	ISO 8601
4:05	ISO 8601
40506	ISO 8601
4:05 AM	与04:05一样，AM不影响数值
4:05 PM	与16:05一样，输入小时数必须<= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	缩写的时区

表 4-13 时区输入

例子	描述
PST	太平洋标准时间（Pacific Standard Time）
-8:00	ISO 8601与PST的偏移
-800	ISO 8601与PST的偏移
-8	ISO 8601与PST的偏移

示例：

```
SELECT time '04:05:06';
   time
```

```
----------
 04:05:06
(1 row)

SELECT time '04:05:06 PST';
   time
----------
 04:05:06
(1 row)

SELECT time with time zone '04:05:06 PST';
   timetz
-------------
 04:05:06-08
(1 row)
```

特殊值

GaussDB(DWS)支持几个特殊值，在读取的时候将被转换成普通的日期/时间值，请参考表4-14。

表 4-14 特殊值

输入字符串	适用类型	描述
epoch	date，timestamp	1970-01-01 00:00:00+00 （Unix系统零时）
infinity	timestamp	比任何其他时间戳都晚
-infinity	timestamp	比任何其他时间戳都早
now	date，time，timestamp	当前事务的开始时间
today	date，timestamp	今日午夜
tomorrow	date，timestamp	明日午夜
yesterday	date，timestamp	昨日午夜
allballs	time	00:00:00.00 UTC

时间段输入

reltime的输入方式可以采用任何合法的时间段文本格式，包括数字形式（含负数和小数）及时间形式，其中时间形式的输入支持SQL标准格式、ISO-8601格式、POSTGRES格式等。另外，文本输入需要加单引号。

时间段输入的详细信息请参考表6 时间段输入。

表 4-15 时间段输入

输入示例	输出结果	描述
60	2 mons	采用数字表示时间段，默认单位是day，可以是小数或负数。特别的，负数时间段，在语义上，可以理解为“早于多久”。
31.25	1 mons 1 days 06:00:00	
-365	-12 mons -5 days	
1 years 1 mons 8 days 12:00:00	1 years 1 mons 8 days 12:00:00	采用POSTGRES格式表示时间段，可以正负混用，不区分大小写，输出结果为将输入时间段计算并转换得到的简化POSTGRES格式时间段。
-13 months -10 hours	-1 years -25 days -04:00:00	
-2 YEARS +5 MONTHS 10 DAYS	-1 years -6 mons -25 days -06:00:00	
P-1.1Y10M	-3 mons -5 days -06:00:00	采用ISO-8601格式表示时间段，可以正负混用，不区分大小写，输出结果为将输入时间段计算并转换得到的简化POSTGRES格式时间段。
-12H	-12:00:00	

示例：

```
--创建表。
CREATE TABLE reltime_type_tab(col1 character(30), col2 reltime);

--插入数据。
INSERT INTO reltime_type_tab VALUES ('90', '90');
INSERT INTO reltime_type_tab VALUES ('-366', '-366');
INSERT INTO reltime_type_tab VALUES ('1975.25', '1975.25');
INSERT INTO reltime_type_tab VALUES ('-2 YEARS +5 MONTHS 10 DAYS', '-2 YEARS +5 MONTHS 10 DAYS');
INSERT INTO reltime_type_tab VALUES ('30 DAYS 12:00:00', '30 DAYS 12:00:00');
INSERT INTO reltime_type_tab VALUES ('P-1.1Y10M', 'P-1.1Y10M');

--查看数据。
SELECT * FROM reltime_type_tab;
             col1              |               col2
--------------------------------+----------------------------------
 1975.25                        | 5 years 4 mons 29 days
 -2 YEARS +5 MONTHS 10 DAYS     | -1 years -6 mons -25 days -06:00:00
 P-1.1Y10M                      | -3 mons -5 days -06:00:00
 -366                           | -1 years -18:00:00
 90                             | 3 mons
 30 DAYS 12:00:00               | 1 mon 12:00:00
(6 rows)

--删除表。
DROP TABLE reltime_type_tab;
```

4.7 几何类型

GaussDB(DWS)支持的几何类型请参见表4-16。最基本的类型：点，是其它类型的基础。

表 4-16 几何类型

名字	存储空间	说明	表现形式
point	16字节	平面中的点	(x,y)
lseg	32字节	（有限）线段	((x1,y1),(x2,y2))
box	32字节	矩形	((x1,y1),(x2,y2))
path	16+16n字节	闭合路径（与多边形类似）	((x1,y1),...)
path	16+16n字节	开放路径	[(x1,y1),...]
polygon	40+16n字节	多边形（与闭合路径相似）	((x1,y1),...)
circle	24 字节	圆	<(x,y),r>（圆心和半径）

GaussDB(DWS)提供了一系列的函数和操作符用来进行各种几何计算，如拉伸、转换、旋转、计算相交等。详细信息请参考**几何函数和操作符**。

点

点是几何类型的基本二维构造单位。用下面语法描述point的数值：

```
( x , y )
x , y
```

x和y是用浮点数表示的点的坐标。

点输出使用第一种语法。

线段

线段（lseg）是用一对点来代表的。用下面的语法描述lseg的数值：

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1   ,   x2 , y2
```

(x1,y1)和(x2,y2)表示线段的端点。

线段输出使用第一种语法。

矩形

矩形是用一对对角点来表示的。用下面的语法描述box的值：

```
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1   ,   x2 , y2
```

(x1,y1)和(x2,y2)表示矩形的一对对角点。

矩形的输出使用第二种语法。

任何两个对角都可以出现在输入中，但按照那样的顺序，右上角和左下角的值会被重新排序以存储。

路径

路径由一系列连接的点组成。路径可能是开放的，也就是认为列表中第一个点和最后一个点没有连接，也可能是闭合的，这时认为第一个和最后一个点连接起来。

用下面的语法描述path的数值：

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1   , ... ,  xn , yn )
x1 , y1   , ... ,  xn , yn
```

点表示组成路径的线段的端点。方括弧（ [] ）表明一个开放的路径，圆括弧（ () ）表明一个闭合的路径。当最外层的括号被省略，如在第三至第五语法，会假定一个封闭的路径。

路径的输出使用第一种或第二种语法输出。

多边形

多边形由一系列点代表（多边形的顶点）。多边形可以认为与闭合路径一样，但是存储方式不一样而且有自己的一套支持函数。

用下面的语法描述polygon的数值：

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1   , ... ,  xn , yn )
x1 , y1   , ... ,  xn , yn
```

点表示多边形的端点。

多边形输出使用第一种语法。

圆

圆由一个圆心和半径标识。用下面的语法描述circle的数值：

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y   , r
```

(x,y)表示圆心，r表示半径。

圆的输出用第一种格式。

4.8 数组类型

数组是一组数据的集合。数组类型允许在单个数据库字段中存储多个值。数组类型通常用于存储和处理具有相似属性的数据。

语法格式

```
ARRAY [ param ]
```

或

```
'{ param }'
```

其中参数param说明如下：

- param ： 数组包含的值，允许出现零个或多个，多个值之间用逗号分隔，没有值可填写为NULL。
- 以'{ param }' 这种格式作为数组常量时，其中的字符串类型的元素不能再以单引号开始和结束，需要使用双引号。两个连续单引号转换为一个单引号。
- 以第一个元素的数据类型作为数组的数据类型，要求数组中所有元素的类型相同，或者能够相互转换。

数组类型的定义

一个数组数据类型可以通过在数组元素的数据类型名称后面加上方括号（ [] ）来命名。

例如，创建表books，其中表示书本价格的列price的类型为一维integer类型数组，表示书本标签的列tag的类型为二维text类型数组。

```
CREATE TABLE books (id SERIAL PRIMARY KEY, title VARCHAR(100), price_by_quarter int[], tags TEXT[][]);
```

CREATE TABLE语法可以指定数组的大小，例如：

```
CREATE TABLE test ( a int[3]);
```

当前的数据库实现会忽略语句中数组的大小限制，即其行为与未指定长度的数组相同。同时，也不会强制所声明的维度数。一个特定元素类型的数组全部被当作是相同的类型，而忽略其大小或维度数。

也可以使用关键词ARRAY来定义一维数组。表books中的列price使用ARRAY定义并指定数组大小，如下所示：

```
price_by_quarter int ARRAY[4]
```

使用ARRAY定义，不指定数组尺寸：

```
price_by_quarter int ARRAY
```

数组值输入

输入数组值时要把一个数组值写成一个文字常数，将元素值用花括号包围并用逗号分隔。一个数组常量的一般格式如下：

```
'{ val1 delim val2 delim ... }'
```

其中，delim是类型的定界符，每个val可以是数组元素类型的一个常量或子数组。

一个数组常量的例子如下：

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

该常量是一个二维的，3乘3数组，它由3个整数子数组构成。

向表books插入数据并查询表books：

```
INSERT INTO books
  VALUES (1, 'One Hundred years of Solitude','{25,25,25,25}','{{"fiction"}, {"adventure"}}'),
         (2, 'Robinson Crusoe', '{30,32,32,32}', '{{"adventure"}, {"fiction"}}'),
         (3, 'Gone with the Wind', '{27,27,29,28}', '{{"romance"}, {"fantasy"}}');

SELECT * FROM books;
 id |            title            | price_by_quarter |          tags
----+-----------------------------+------------------+-------------------------
  1 | One Hundred years of Solitude | {25,25,25,25}    | {{fiction},{adventure}}
  2 | Robinson Crusoe               | {30,32,32,32}    | {{adventure},{fiction}}
  3 | Gone with the Wind            | {27,27,29,28}    | {{romance},{fantasy}}
(3 rows)
```

⚠ 注意

插入多维数组数据时，多维数组的每一维都必须有相匹配的长度。

使用ARRAY关键字插入数据：

```
INSERT INTO books
  VALUES (1, 'One Hundred years of Solitude',ARRAY[25,25,25,25],ARRAY['fiction', 'adventure']),
         (2, 'Robinson Crusoe', ARRAY[30,32,32,32], ARRAY['adventure', 'fiction']),
         (3, 'Gone with the Wind', ARRAY[27,27,29,28], ARRAY['romance', 'fantasy']);
```

访问数组

访问数组元素

查询在第二季度价格发生变化的书籍的名称：

```
SELECT title FROM books WHERE price_by_quarter[1] <> price_by_quarter[2];
      title
-----------------
 Robinson Crusoe
(1 row)
```

查询检索所有书籍第三季度的价格：

```
SELECT price_by_quarter[3] FROM books;
 price_by_quarter
------------------
               29
               25
               32
(3 rows)
```

访问数组切片

可以访问一个数组的任意矩形切片或者子数组。一个数组切片可以通过在一个或多个数组维度上指定[下界:上界]来定义。

查询Gone with the Wind的第二个标签：

```
SELECT tags[2:2] FROM books WHERE title = 'Gone with the Wind';
   tags
-----------
 {fantasy}
(1 row)
```

使用函数访问数组

使用array_dims函数获得数组值的当前维度：

```
SELECT array_dims(tags) FROM books WHERE title = 'Robinson Crusoe';
 array_dims
------------
 [1:2]
(1 row)
```

也可以使用array_upper和array_lower来获得数组维度，它们将分别返回一个指定数组的上界和下界：

```
SELECT array_upper(tags, 1) FROM books WHERE title = 'Robinson Crusoe';
 array_upper
-------------
           2
(1 row)
```

array_length函数将返回一个指定数组维度的长度：

```
SELECT array_length(tags, 1) FROM books WHERE title = 'Robinson Crusoe';
 array_length
--------------
            2
(1 row)
```

修改数组

更新数组

更新整个数组数据：

```
UPDATE books SET price_by_quarter = '{30,30,30,30}'
    WHERE title = 'Robinson Crusoe';
```

使用ARRAY表达式语法更新整个数组数据：

```
UPDATE books SET price_by_quarter = ARRAY[30,30,30,30]
    WHERE title = 'Robinson Crusoe';
```

更新数组中的一个元素：

```
UPDATE books SET price_by_quarter[4] = 35
    WHERE title = 'Robinson Crusoe';
```

更新数组中的一个切片元素：

```
UPDATE books SET price_by_quarter[1:2] = '{27,27}'
    WHERE title = 'Robinson Crusoe';
```

一个已存储的数组值可以被通过对其还不存在的元素赋值来扩大大小。任何位于已存在的元素和新元素之间的位置都将被空值填充。例如，如果数组myarray目前有4个元素，使用UPDATE对myarray[6]赋值后它将有6个元素，其中myarray[5]为空值。目前，采用这种方式扩大数组只允许使用在一维数组上。

构建新数组

新的数组值也可以通过串接操作符“||”构建。串接操作符允许把一个单独的元素加入到一个一维数组的开头或末尾。也可接受两个N维数组，或者一个N维数组和一个N+1维数组。

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
     ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

使用函数array_prepend、array_append或array_cat构建数组。

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)
```



```
SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);

     array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
     array_cat
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

4.9 枚举类型

枚举（enum）类型是由一个静态、值的有序集合构成的数据类型。它们等效于很多编程语言所支持的enum类型。枚举类型可以是一周中的日期，或者一个数据的状态值集合。

枚举类型的声明

枚举类型可以使用CREATE TYPE命令创建，例如：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

枚举类型被创建后，可以在表和函数定义中使用：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (name text, current_mood mood);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

排序

枚举类型值的排序是该类型被创建时所列出的值的顺序。

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');

SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name FROM person WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
```



```
-------
 Larry
(1 row)
```

枚举类型的安全性

每一种枚举数据类型都是独立的并且不能和其他枚举类型相比较。

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (num_weeks integer, happiness happiness);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');

INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"

SELECT person.name, holidays.num_weeks FROM person, holidays
      WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

如果需要作比较，可以使用自定义的操作符或者在查询中加上显式类型：

```
 SELECT person.name, holidays.num_weeks FROM person, holidays
      WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

注意事项

- 枚举标签是大小写敏感的，因此'happy'与'HAPPY'是不同的。标签中的空格也是有意义的。
- 尽管枚举类型的主要目的是用于值的静态集合，但也有方法在现有枚举类型中增加新值和重命名值（见**ALTER TYPE**）。不能从枚举类型中去除现有的值，也不能更改这些值的排序顺序，除非删除并且重建枚举类型。
- 从内部枚举值到文本标签的翻译保存在系统表PG_ENUM中。

4.10 网络地址类型

GaussDB(DWS)提供用于存储IPv4、IPv6、MAC地址的数据类型。

网络地址类型提供输入错误检查和特殊的操作和功能（请参见**网络地址函数和操作符**），比纯文本类型更适合存储IPv4、IPv6、MAC地址的数据类型。

表 4-17 网络地址类型

名字	存储空间	描述
cidr	7或19字节	IPv4或IPv6网络
inet	7或19字节	IPv4或IPv6主机和网络
macaddr	6字节	MAC地址

在对inet或cidr数据类型进行排序的时候，IPv4地址总是排在IPv6地址前面，包括那些封装或者是映射在IPv6地址里的IPv4地址，例如::10.2.3.4或::ffff:10.4.3.2。

cidr

cidr（无类别域间路由，Classless Inter-Domain Routing）类型，保存一个IPv4或IPv6网络地址。声明网络格式为address/y，address表示IPv4或者IPv6地址，y表示子网掩码的二进制位数。如果省略y，则掩码部分使用已有类别的网络编号系统进行计算，但要求输入的数据已经包括了确定掩码所需的所有字节。

- 示例一：CIDR格式换算为IP地址网段

  例如，10.0.0.0/8换算为32位二进制地址：00001010.00000000.00000000.00000000。其中/8表示8位网络ID，即32位二进制地址中前8位是固定不变的，对应网段为：00001010.00000000.00000000.00000000~00001010.11111111.11111111.11111111。则换算为十进制后，10.0.0.0/8表示：子网掩码为255.0.0.0，对应网段为10.0.0.0~10.255.255.255。

- 示例二：IP地址网段换算为CIDR格式

  例如，192.168.0.0~192.168.31.255，后两段IP换算为二进制地址：00000000.00000000~00011111.11111111，可以得出前19位（8*2+3）是固定不变的，则换算为CIDR格式后，表示为：192.168.0.0/19。

表 4-18 cidr 类型输入举例

cidr输入	cidr输出	abbrev（cidr）
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1

inet

inet类型在一个数据区域内保存主机的IPv4或IPv6地址，以及一个可选子网。主机地址中网络地址的位数表示子网（“子网掩码”）。如果子网掩码是32并且地址是IPv4，

则这个值不表示任何子网，只表示一台主机。在IPv6里，地址长度是128位，因此128位表示唯一的主机地址。

该类型的输入格式是address/y，address表示IPv4或者IPv6地址，y是子网掩码的二进制位数。如果省略/y，则子网掩码对IPv4是32，对IPv6是128，所以该值表示只有一台主机。如果该值表示只有一台主机，/y将不会显示。

inet和cidr类型之间的基本区别是inet接受子网掩码，而cidr不接受。

macaddr

macaddr类型存储MAC地址，也就是以太网卡硬件地址（尽管MAC地址还用于其它用途）。可以接受下列格式：

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08002b010203'
```

这些示例都表示同一个地址。对于数据位a到f，大小写都行。输出时都是以第一种形式展示。

4.11 位串类型

位串就是一串1和0的字符串。它们可以用于存储位掩码。

GaussDB(DWS)支持两种位串类型：bit(n)和bit varying(n)，其中n是一个正整数。

bit类型的数据必须准确匹配长度n，如果存储短或者长的数据都会报错。bit varying类型的数据是最长为n的变长类型，超过n的类型会被拒绝。一个没有长度的bit等效于bit(1)，没有长度的bit varying表示没有长度限制。

说明

如果显式地把一个位串值转换成bit(n)，则此位串右边的内容将被截断或者在右边补齐零，直到刚好n位，而且不会抛出任何错误。类似地，如果显式地把一个位串数值转换成bit varying(n)，如果它超过了n位，则它的右边将被截断。

位串类型使用示例：

1. 创建示例表bit_type_t1：
   ```
   CREATE TABLE bit_type_t1
   (
      BT_COL1 INTEGER,
      BT_COL2 BIT(3),
      BT_COL3 BIT VARYING(5)
   ) DISTRIBUTE BY REPLICATION;
   ```
2. 插入数据：
   ```
   INSERT INTO bit_type_t1 VALUES(1, B'101', B'00');
   ```
   插入数据的长度不符合类型的标准会报错。
   ```
   INSERT INTO bit_type_t1 VALUES(2, B'10', B'101');
   ERROR:  bit string length 2 does not match type bit(3)
   CONTEXT:  referenced column: bt_col2
   ```
3. 将不符合类型长度的数据进行转换：
   ```
   INSERT INTO bit_type_t1 VALUES(2, B'10'::bit(3), B'101');
   ```
4. 查看数据：

```
SELECT * FROM bit_type_t1;
 bt_col1 | bt_col2 | bt_col3
---------+---------+---------
       1 | 101     | 00
       2 | 100     | 101
(2 rows)
```

4.12 文本搜索类型

GaussDB(DWS)提供了tsvector和tsquery两种数据类型用于支持全文检索。tsvector类型表示为文本搜索优化的文件格式，tsquery类型表示文本查询。

tsvector

tsvector类型表示一个检索单元，通常是一个数据库表中的一行文本字段或者这些字段的组合。

tsvector类型的值是唯一分词的分类列表，把一句话的词格式化为不同的词条，在进行分词处理的时候tsvector会按照一定的顺序录入，并自动去掉分词中重复的词条。

to_tsvector函数通常用于解析和标准化文档字符串。

通过tsvector把一个字符串按照空格进行分词，分词的顺序是按照字母和长短排序的，请看以下例子：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                    tsvector
-----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
(1 row)
```

如果词条中包含空格或标点符号，可以用引号包围：

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
(1 row)
```

使用常规的单引号引起来的字符串，字符串中嵌入的单引号(')和反斜杠(\)必须双写进行转义：

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                  tsvector
-------------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
(1 row)
```

词条位置常量也可以放到词汇中：

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
(1 row)
```

位置常量通常表示文档中源字的位置。位置信息可以用于进行排名。位置常量的范围是1到16383，最大值默认是16383。相同词的重复位会被忽略掉。

拥有位置的词汇可以进一步地被标注一个权重，它可以是A，B，C或D。 D是默认权重，因此输出中不会显示：

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
         tsvector
```

```
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
(1 row)
```

权重通常被用来反映文档结构，如：将标题标记成与正文词不同。文本检索排序函数可以为不同的权重标记分配不同的优先级。

tsvector类型标准用法示例如下：

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
(1 row)
```

但是对于英文全文检索应用来说，上面的单词会被认为非规范化的，所以需要通过to_tsvector函数对这些单词进行规范化处理：

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
(1 row)
```

tsquery

tsquery类型表示一个检索条件，存储用于检索的词汇，并且使用布尔操作符&（AND），|（OR）和!（NOT）将这些词汇进行组合，圆括号用来强调操作符的分组。如果没有圆括号，（NOT）的优先级最高，其次是&（AND），最后是|（OR）。to_tsquery函数及plainto_tsquery函数会将单词转换为tsquery类型前进行规范化处理。

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'
(1 row)

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )
(1 row)

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
(1 row)
```

tsquery中的词汇可以用一个或多个权重字母来标记，这些权重字母限制词汇只能与匹配权重的tsvector词汇进行匹配。

```
SELECT 'fat:ab & cat'::tsquery;
     tsquery
------------------
 'fat':AB & 'cat'
(1 row)
```

同样，tsquery中的词汇可以用*标记来指定前缀匹配。例如：这个查询可以匹配tsvector中以“super”开始的任意单词。

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
```



```
 'super':*
(1 row)
```

需注意，前缀匹配会首先被文本搜索分词器处理。例如：postgres中提取的词干是postgr，匹配到了postgraduate，也就意味着下面的结果为真：

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' ) AS RESULT;
 result
----------
 t
(1 row)
SELECT to_tsquery('postgres:*');
 to_tsquery
------------
 'postgr':*
(1 row)
```

to_tsquery函数会将单词转换为tsquery类型前进行规范化处理。'Fat:ab & Cats'规范化转为tsquery类型结果如下：

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
(1 row)
```

4.13 UUID 类型

UUID：通用唯一识别码（Universally Unique Identifier）是用于计算机体系中以识别信息的一个128位标识符。

UUID的作用是让分布式系统中的所有元素都能有唯一的辨识信息，而不需要通过中央控制端来做辨识信息的指定。很多应用场景需要一个ID，仅用来标识一个对象。常见的例子有数据库表的ID字段。另一个例子是前端的各种UI库，因为它们通常需要动态创建各种UI元素，这些元素需要唯一的ID，这时就需要使用UUID。

GaussDB(DWS)支持的UUID函数及UUID应用示例，可参考**UUID函数**。

UUID 格式

UUID由开放软件基金会标准化，作为分布式计算环境的一部分，在互联网工程任务组（IETF）公布的RFC 4122标准中对UUID进行了标准化。标准的UUID由36个字符组成，其中包括32个16进制数字和4个连字符‘-’，形式为8-4-4-4-12，标准的UUID示例如下：

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

除了标准型的UUID，GaussDB(DWS)同样支持以其他方式输入：大写字母和数字、由花括号包围的标准格式、省略部分或所有连字符、在任意一组四位数字之后加一个连字符。示例：

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
```

UUID 组成

UUID采用时间信息和空间信息确保全局唯一，通常由时间戳信息、时钟序列和节点ID（MAC地址）组成。而GaussDB(DWS)分布式集群中多个节点可能部署在同一个机器上，其MAC地址相同，UUID存在冲突的风险。因此GaussDB(DWS)将最后48位为的

MAC地址替换为生成UUID的CN或DN的序号和当前的线程ID，确保UUID在分布式集群内部做到全局唯一。

4.14 JSON 类型

JSON数据类型可以用来存储JSON（JavaScript Object Notation）数据。

可以是单独的一个标量，也可以是一个数组，也可以是一个键值对象，其中数组和对象可以统称容器(container)：

1. 标量(scalar)：单一的数字、bool、string、null都可以叫做标量。
2. 数组(array)：[]结构，里面存放的元素可以是任意类型的JSON，并且不要求数组内所有元素都是同一类型。
3. 对象(object)：{}结构，存储key:value的键值对，其键只能是用“ ”包裹起来的字符串，值可以是任意类型的JSON，对于重复的键，按最后一个键值对为准。

GaussDB(DWS)支持使用：json数据类型和jsonb数据类型存储JSON数据。其中：

- json对输入的字符串进行完整复制，使用时再去解析，所以它会保留输入的空格，重复键以及顺序等。
- jsonb解析输入后保存的二进制，它在解析时会删除语义无关的细节和重复的键，对键值也会进行排序，使用时不用再次解析。

因此可以发现，两者其实都是JSON，它们接受相同的字符串作为输入。它们实际的主要差别是效率。json数据类型存储输入文本的精确复制，处理函数必须在每个执行上重新解析； 而jsonb数据以分解的二进制格式存储， 这使得它由于添加了转换机制而在输入上稍微慢些，但是在处理上明显更快， 因为不需要重新解析。同时由于jsonb类型存在解析后的格式归一化等操作，同等的语义下只会有一种格式，因此可以更好更强大的支持很多其他额外的操作，比如按照一定的规则进行大小比较等。jsonb也支持索引，这也是一个明显的优势。

输入格式

json和jsonb输入必须是一个符合JSON数据格式的字符串，此字符串用单引号''声明。

null (null-json)：仅null，全小写。

```
SELECT 'null'::json;   -- suc
SELECT 'NULL'::jsonb;  -- err
```

数字 (num-json)：正负整数、小数、0，支持科学计数法。

```
SELECT '1'::json;
SELECT '-1.5'::json;
SELECT '-1.5e-5'::jsonb, '-1.5e+2'::jsonb;
SELECT '001'::json, '+15'::json, 'NaN'::json;  -- 不支持多余的前导0，正数的+号，以及NaN和infinity。
```

布尔(bool-json)：仅true、false，全小写。

```
SELECT 'true'::json;
SELECT 'false'::jsonb;
```

字符串(str-json)：必须是加双引号的字符串。

```
SELECT '"a"'::json;
SELECT '"abc"'::jsonb;
```

数组(array-json)：使用中括号[]包裹，满足数组书写条件。数组内元素类型可以是任意合法的JSON，且不要求类型一致。

```
SELECT '[1, 2, "foo", null]'::json;
SELECT '[]'::json;
SELECT '[1, 2, "foo", null, [[]], {}]'::jsonb;
```

对象(object-json)：使用大括号{}包裹，键必须是满足JSON字符串规则的字符串，值可以是任意合法的JSON。

```
SELECT '{}'::json;
SELECT '{"a": 1, "b": {"a": 2,  "b": null}}'::json;
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::jsonb;
```

⚠ 注意

- 'null'::json 和 null::json 是两个不同的概念，类似于字符串str=“ ”和str=null。
- 对于数字，当使用科学计数法的时候，jsonb类型会将其展开，而json会精准复制输入。

jsonb 高级特性

json和jsonb的主要差异在于存储方式上的不同，jsonb存储的是解析后的二进制，能够体现JSON的层次结构，更便于直接访问等，因此jsonb较json具有很多高级特性。

格式归一化

- 对于输入的object-json字符串，解析成jsonb二进制后，会天然的丢弃语义上无关紧要的细节，比如空格：

```
SELECT '    [1, " a ", {"a"    :1     }]   '::jsonb;
    jsonb
----------------------
[1, " a ", {"a": 1}]
(1 row)
```

- 对于object-json，会删除重复的键值，只保留最后一个出现的，例如：

```
SELECT '{"a" : 1, "a" : 2}'::jsonb;
jsonb
----------
{"a": 2}
(1 row)
```

- 对于object-json，键值会重新进行排序，排序规则：长度长的在后、长度相等则ascii码大的在后，例如：

```
SELECT '{"aa" : 1, "b" : 2, "a" : 3}'::jsonb;
       jsonb
----------------------------
{"a": 3, "b": 2, "aa": 1}
(1 row)
```

大小比较

由于经过了格式归一化，保证了同一种语义下的jsonb只会有一种存在形式，因此按照制定的规则，可以比较大小。

1. 首先比较类型：object-jsonb > array-jsonb > bool-jsonb > num-jsonb > str-jsonb > null-jsonb
2. 同类型则比较内容：
   - str-json类型：依据text比较的方法，使用数据库默认排序规则进行比较，返回值正数代表大于，负数代表小于，0表示相等。
   - num-json类型：数值比较

- bool-json类型：true > false
- array-jsonb类型：长度长的 > 长度短的，长度相等则依次比较每个元素。
- object-jsonb类型：长度长的 > 长度短的，长度相等则依次比较每个键值对，先比较键，再比较值。

⚠ 注意

object-jsonb类型内比较时使用的是格式整理后的最终结果进行比较，因此相对于直接的输入未必会很直观。

创建索引

jsonb类型支持创建btree索引和GIN索引。

如果整个JSONB列使用Btree索引，则能使用的运算符是=、<、<=、>、>=。

示例：创建表test，并插入数据。

```
CREATE TABLE test(id bigserial, data JSONB, PRIMARY KEY (id));
INSERT INTO test(data) VALUES('{"name":"Jack", "age":10, "nick_name":["Jacky","baobao"], "phone_list":["1111","2222"]}'::jsonb);
```

- 创建Btree索引
  ```
  CREATE INDEX idx_test_data_age ON test USING btree(((data->>'age')::int));
  ```
  使用Btree索引查询age>1
  ```
  SELECT * FROM test WHERE (data->>'age')::int>1;
  ```
- 创建GIN索引
  ```
  CREATE INDEX idx_test_data ON test USING gin (data);
  ```
  使用GIN索引查询顶层关键词是否存在
  ```
  SELECT * FROM test WHERE data ? 'id';
  SELECT * FROM test WHERE data ?| array['id','name'];
  ```
- 使用GIN索引查询非顶层关键词是否存在
  ```
  CREATE INDEX idx_test_data_nick_name ON test USING gin((data->'nick_name'));
  SELECT * FROM test WHERE data->'nick_name' ? 'Jacky';
  ```
- 使用@>查询JSON中是否包含子json对象
  ```
  SELECT * FROM test WHERE data @> '{"age":10, "nick_name":["Jacky"]}';
  ```

包含存在

查询一个JSON之中是否包含某些元素，或者某些元素是否存在于某个JSON中是jsonb的一个重要能力。

- 简单的标量/原始值只包含相同的值
  ```
  SELECT '"foo"'::jsonb @> '"foo"'::jsonb;
  ```
- 左侧数组包含了右侧字符串。
  ```
  SELECT '[1, "aa", 3]'::jsonb ? 'aa';
  ```
- 左侧数组包含了右侧的数组所有元素，顺序、重复不重要。
  ```
  SELECT '[1, 2, 3]'::jsonb @> '[1, 3, 1]'::jsonb;
  ```
- 左侧object-json包含了右侧object-json的所有键值。
  ```
  SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb":true}'::jsonb @> '{"version":9.4}'::jsonb;
  ```
- 左侧数组并没有包含右侧的数组所有元素，因为左侧数组的三个元素为1、2、[1,3]，右侧的为1、3
  ```
  SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb; -- false
  ```
- 相似的这种也没有包含。

```
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb; -- false
```

4.15 RoaringBitmap 类型

GaussDB(DWS)自8.1.3集群版本开始，支持RoaringBitmap数据类型，用于存储位图数据集。

roaringbitmap数据类型支持行存，列存表。

表 4-19 RoaringBitmap 类型

名字	存储容量	描述	范围
RoaringBitmap	32 字节	存储位图数据集	-2,147,483,648~2,147,483,647

示例：创建带有roaringbitmap数据类型的表。

```
CREATE TABLE r_row (a int ,b text, c roaringbitmap);
CREATE TABLE r_col (a int ,b text, c roaringbitmap) with (orientation=column);
```

4.16 HLL 数据类型

HLL（HyperLoglog）是统计数据集中唯一值个数的高效近似算法。它有着计算速度快，节省空间的特点，不需要直接存储集合本身，而是存储一种名为HLL的数据结构。每当有新数据加入进行统计时，只需要把数据经过哈希计算并插入到HLL中，最后根据HLL就可以得到结果。

HLL与其他算法的比较请参见**表4-20**。

表 4-20 HLL 与其他算法比较

项目	Sort算法	Hash算法	HLL
时间复杂度	O(nlogn)	O(n)	O(n)
空间复杂度	O(n)	O(n)	1280 bytes
误差率	0	0	≈2%
所需存储空间	原始数据大小	原始数据大小	1280 bytes

HLL在计算速度和所占存储空间上都占优势。在时间复杂度上，Sort算法需要排序至少O(nlogn)的时间，虽说Hash算法和HLL一样扫描一次全表O(n)的时间就可以得出结果，但是存储空间上， Sort算法和Hash算法都需要先把原始数据存起来再进行统计，会导致存储空间消耗巨大，而对HLL来说不需要存原始数据，只需要维护HLL数据结构，故占用空间始终是1280bytes常数级别。

须知

- 当前默认规格下可计算最大distinct值的数量为1.6e+12个，误差率最大仅2.3%。用户应注意如果计算结果超过当前规格下distinct最大值会导致计算结果误差率变大，或导致计算结果失败并报错。
- 用户在首次使用该特性时，应该对业务的distinct value做评估，选取适当的配置参数并做验证，以确保精度符合要求：
  - 当前默认参数下，可以计算的distinct value值为1.6e+12，如果计算得到的distinct value值为NaN，需要调整log2m和regwidth来容纳更多的distinct value。
  - 虽然hash算法存在极低的hash collision概率，但是建议用户在首次使用时，选取2-3个hash seed验证，如果得到的distinct value相差不大，则可以从该组seed中任选一个作为hash seed。

HLL中主要的数据结构，请参见**表4-21**。

表 4-21 HyperLogLog 中主要数据结构

数据类型	功能描述
hll	大小为确定的1280 bytes，可直接计算得到distinct值。

HLL 的应用场景

- 使用hll数据类型场景
  a. 创建带有hll类型的表并向表中插入空的hll。
  ```
  CREATE TABLE helloworld (id integer, set hll);
  INSERT INTO helloworld(id, set) VALUES (1, hll_empty());
  ```
  b. 把整数经过哈希计算加入到hll中。
  ```
  UPDATE helloworld SET set = hll_add(set, hll_hash_integer(12345)) WHERE id = 1;
  ```
  c. 把字符串经过哈希计算加入到hll中。
  ```
  UPDATE helloworld SET set = hll_add(set, hll_hash_text('hello world')) WHERE id = 1;
  ```
  d. 得到hll中的distinct值。
  ```
  SELECT hll_cardinality(set) FROM helloworld WHERE id = 1;
   hll_cardinality
  -----------------
                2
  (1 row)
  ```
- 使用hll进行网站访客统计场景
  a. 创建原始数据表facts，记录用户访问网站时间。
  ```
  CREATE TABLE facts (
          date            date,
          user_id         integer
  );
  ```
  b. 插入用户访问过网站的数据。
  ```
  INSERT INTO facts VALUES ('2019-02-20', generate_series(1,100));
  INSERT INTO facts VALUES ('2019-02-21', generate_series(1,200));
  INSERT INTO facts VALUES ('2019-02-22', generate_series(1,300));
  INSERT INTO facts VALUES ('2019-02-23', generate_series(1,400));
  INSERT INTO facts VALUES ('2019-02-24', generate_series(1,500));
  INSERT INTO facts VALUES ('2019-02-25', generate_series(1,600));
  ```

```
INSERT INTO facts VALUES ('2019-02-26', generate_series(1,700));
INSERT INTO facts VALUES ('2019-02-27', generate_series(1,800));
```

c. 创建表并指定列为hll。根据日期把数据分组，并把数据插入到hll中。

```
CREATE TABLE daily_uniques (
    date            date UNIQUE,
    users           hll
);

INSERT INTO daily_uniques(date, users)
    SELECT date, hll_add_agg(hll_hash_integer(user_id))
    FROM facts
    GROUP BY 1;
```

d. 计算每一天访问网站不同用户数量。

```
SELECT date, hll_cardinality(users) FROM daily_uniques ORDER BY date;
        date         | hll_cardinality
---------------------+------------------
 2019-02-20 00:00:00 |              100
 2019-02-21 00:00:00 | 203.813355588808
 2019-02-22 00:00:00 | 308.048239950384
 2019-02-23 00:00:00 | 410.529188080374
 2019-02-24 00:00:00 | 513.263875705319
 2019-02-25 00:00:00 | 609.271181107416
 2019-02-26 00:00:00 | 702.941844662509
 2019-02-27 00:00:00 | 792.249946595237
(8 rows)
```

e. 计算在2019.02.20到2019.02.26一周中有多少不同用户访问过网站。

```
SELECT hll_cardinality(hll_union_agg(users)) FROM daily_uniques WHERE date >=
'2019-02-20'::date AND date <= '2019-02-26'::date;
 hll_cardinality
------------------
 702.941844662509
(1 row)
```

f. 计算昨天访问过网站而今天没访问网站的用户数量。

```
SELECT date, (#hll_union_agg(users) OVER two_days) - #users AS lost_uniques FROM
daily_uniques WINDOW two_days AS (ORDER BY date ASC ROWS 1
PRECEDING);
        date         | lost_uniques
---------------------+--------------
 2019-02-20 00:00:00 |            0
 2019-02-21 00:00:00 |            0
 2019-02-22 00:00:00 |            0
 2019-02-23 00:00:00 |            0
 2019-02-24 00:00:00 |            0
 2019-02-25 00:00:00 |            0
 2019-02-26 00:00:00 |            0
 2019-02-27 00:00:00 |            0
(8 rows)
```

- 插入数据不满足hll数据结构要求时报错场景

  当用户给hll类型的字段插入数据的时候，必须保证插入的数据满足hll数据结构要求，如果解析后不满足就会报错。

  例如： 插入数据'E\\1234'时，该数据不满足hll数据结构，不能解析成功因此失败报错。

```
CREATE TABLE test(id integer, set hll);
INSERT INTO test VALUES(1, 'E\\1234');
ERROR:  invalid input syntax for integer: "E\\1234"
```

4.17 对象标识符类型

GaussDB(DWS)在内部使用对象标识符（OID）作为各种系统表的主键。系统不会给用户创建的表增加一个OID系统字段，OID类型代表一个对象标识符。

目前OID类型用一个四字节的无符号整数实现。因此不建议在创建的表中使用OID字段做主键。

表 4-22 对象标识符类型

名称	引用	描述	示例
OID	-	数字化的对象标识符。	564182
CID	-	命令标识符。它是系统字段cmin和cmax的数据类型。命令标识符是32位的量。	-
XID	-	事务标识符。它是系统字段xmin和xmax的数据类型。事务标识符也是32位的量。	-
TID	-	行标识符。它是系统表字段ctid的数据类型。行ID是一对数值（块号，块内的行索引），它标识该行在其所在表内的物理位置。	-
REGCONFIG	pg_ts_config	文本搜索配置。	english
REGDICTIONARY	pg_ts_dict	文本搜索字典。	simple
REGOPER	pg_operator	操作符名。	+
REGOPERATOR	pg_operator	带参数类型的操作符。	*(integer,integer)或-(NONE,integer)
REGPROC	pg_proc	函数名字。	sum
REGPROCEDURE	pg_proc	带参数类型的函数。	sum(int4)
REGCLASS	pg_class	关系名。	pg_type
REGTYPE	pg_type	数据类型名。	integer

OID类型：主要作为数据库系统表中字段使用。

示例：

```
SELECT oid FROM pg_class WHERE relname = 'pg_type';
 oid
------
 1247
(1 row)
```

OID别名类型REGCLASS：主要用于对象OID值的简化查找。

示例：


```
SELECT attrelid,attname,atttypid,attstattarget FROM pg_attribute WHERE attrelid = 'pg_type'::REGCLASS;
 attrelid |     attname     | atttypid | attstattarget
----------+-----------------+----------+---------------
     1247 | xc_node_id      |       23 |             0
     1247 | tableoid        |       26 |             0
     1247 | cmax            |       29 |             0
     1247 | xmax            |       28 |             0
     1247 | cmin            |       29 |             0
     1247 | xmin            |       28 |             0
     1247 | oid             |       26 |             0
     1247 | ctid            |       27 |             0
     1247 | typname         |       19 |            -1
     1247 | typnamespace    |       26 |            -1
     1247 | typowner        |       26 |            -1
     1247 | typlen          |       21 |            -1
     1247 | typbyval        |       16 |            -1
     1247 | typtype         |       18 |            -1
     1247 | typcategory     |       18 |            -1
     1247 | typispreferred  |       16 |            -1
     1247 | typisdefined    |       16 |            -1
     1247 | typdelim        |       18 |            -1
     1247 | typrelid        |       26 |            -1
     1247 | typelem         |       26 |            -1
     1247 | typarray        |       26 |            -1
     1247 | typinput        |       24 |            -1
     1247 | typoutput       |       24 |            -1
     1247 | typreceive      |       24 |            -1
     1247 | typsend         |       24 |            -1
     1247 | typmodin        |       24 |            -1
     1247 | typmodout       |       24 |            -1
     1247 | typanalyze      |       24 |            -1
     1247 | typalign        |       18 |            -1
     1247 | typstorage      |       18 |            -1
     1247 | typnotnull      |       16 |            -1
     1247 | typbasetype     |       26 |            -1
     1247 | typtypmod       |       23 |            -1
     1247 | typndims        |       23 |            -1
     1247 | typcollation    |       26 |            -1
     1247 | typdefaultbin   |      194 |            -1
     1247 | typdefault      |       25 |            -1
     1247 | typacl          |     1034 |            -1
(38 rows)
```

4.18 伪类型

GaussDB(DWS)数据类型中包含一系列特殊用途的类型，这些类型按照类别被称为伪类型。伪类型不能作为字段的数据类型，但是可以用于声明函数的参数或者结果类型。

当一个函数不仅仅是简单地接受并返回某种SQL数据类型，伪类型能起到很大的作用。表4-23列出了所有的伪类型。

表 4-23 伪类型

名字	描述
any	表示函数接受任何输入数据类型。
anyelement	表示函数接受任何数据类型。
anyarray	表示函数接受任意数组数据类型。
anynonarray	表示函数接受任意非数组数据类型。

名字	描述
anyenum	表示函数接受任意枚举数据类型。
anyrange	表示函数接受任意范围数据类型。
cstring	表示函数接受或者返回一个空结尾的C字符串。
internal	表示函数接受或者返回一种服务器内部的数据类型。
language_handler	声明一个过程语言调用句柄返回language_handler。
fdw_handler	声明一个外部数据封装器返回fdw_handler。
record	标识函数返回一个未声明的行类型。
trigger	声明一个触发器函数返回trigger。
void	表示函数不返回数值。
opaque	一个已经过时的类型，以前用于所有上面这些用途。

声明用C编写的函数（不管是内置的还是动态装载的）都可以接受或者返回任何这样的伪数据类型。当伪类型作为参数类型使用时，用户需要保证函数的正常运行。

用过程语言编写的函数只能使用实现语言允许的伪类型。目前，过程语言不允许使用伪类型作为参数类型，只允许使用void和record作为结果类型。一些多态的函数还支持使用anyelement，anyarray，anynonarray anyenum和anyrange类型。

伪类型internal用于声明只能在数据库系统内部调用的函数，这些函数不能直接在SQL查询里调用。如果某函数至少有一个internal类型的参数，则不能从SQL里调用。建议不要创建任何声明返回internal的函数，除非该函数至少有一个internal类型的参数。

示例：

创建或替换函数showall()。

```
CREATE OR REPLACE FUNCTION showall() RETURNS SETOF record
AS $$ SELECT count(*) from tpcds.store_sales where ss_customer_sk = 9692; $$
LANGUAGE SQL;
```

调用函数showall()。

```
SELECT showall();
 showall
---------
 (35)
(1 row)
```

删除函数。

```
DROP FUNCTION showall();
```

4.19 范围类型

范围类型是表示某些元素类型（称为范围的子类型）的值范围的数据类型。例如，时间戳范围可用于表示保留会议室的时间范围。在这种情况下，数据类型为tsrange（“时间戳范围”的缩写），时间戳是子类型。子类型必须具有总体的顺序，以便很好地定义元素值是在值的范围内、之前还是之后。

范围类型可以表达单一范围值中的多个元素值，并且可以很清晰地表达诸如范围重叠等概念。用于计划安排的时间和日期范围，也可以用于价格范围或仪器的测量范围等等。

内置范围类型

GaussDB(DWS)带有下列内置范围类型：

- int4range — integer的范围
- int8range — bigint的范围
- numrange — numeric的范围
- tsrange — 不带时区的 timestamp的范围
- tstzrange — 带时区的 timestamp的范围
- daterange — date的范围

说明

GaussDB(DWS)暂不支持用户自定义范围类型。

创建表reservation并插入数据，其中during字段类型为tsrange：

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');
```

查询范围是否包含：

```
SELECT int4range(10, 20) @> 3;
 ?column?
----------
 f
(1 row)
```

查询范围是否重叠：

```
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);
 ?column?
----------
 t
(1 row)
```

抽取范围的上界：

```
SELECT upper(int8range(15, 25));
 upper
-------
    25
(1 row)
```

计算范围的交集：

```
SELECT int4range(10, 20) * int4range(15, 25);
 ?column?
----------
 [15,20)
(1 row)
```

查询范围是否为空：

```
SELECT isempty(numrange(1, 5));
 isempty
---------
 f
(1 row)
```

包含和排除边界

每一个非空范围都有两个界限：下界和上界。这些值之间的所有点都被包括在范围内。包含界限意味着边界点本身也被包括在范围内，而排除边界意味着边界点不被包括在范围内。

在范围的文本形式中，包含下界用“[”表示，排除下界用“(”表示。同样，包含上界用“]”表示，排除上界用“)”表示。

函数**lower_inc(anyrange)**和**lower_inc(anyrange)**分别测试一个范围值的上下界。

无限（无界）范围

范围的下界可以省略，这意味着所有小于上界的值都包括在范围中，例如(,3]。同样，范围的上界被省略，则所有大于下界的值都包括在范围中。如果下界和上界都被省略，则该元素类型的所有值都被认为在范围内。如果缺失的边界指定为包含则自动将包含转换为排除，例如[,]将转换为(,)。可以将这些缺失的值视为+/-无穷大，但它们是特殊的范围类型值，并且被视为超出任何范围元素类型的+/-无穷大值。

具有“无穷大”概念的元素类型可以将其作为显式边界值。例如，在时间戳范围，[today,infinity)不包括特殊的timestamp值infinity，尽管[today,infinity]包括它, 就好比 [today,)和[today,]。

函数**lower_inf(anyrange)**和**upper_inf(anyrange)**分别测试一个范围的无限上下界。

范围的输入/输出

范围值的输入必须遵循下列模式之一：

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

圆括号或方括号指示上下界是否为排除的或者包含的。最后一个模式是empty，它表示一个空范围（一个不包含点的范围）。

lower-bound可以作为子类型的合法输入的一个字符串，或者是空，表示没有下界。同样，upper-bound可以是作为子类型的合法输入的一个字符串，或者是空，表示没有上界。

每个界限值可以使用双引号引用。如果界限值包含圆括号、方括号、逗号、双引号或反斜线时，则必须使用双引号引用，否则这些符号会被认作范围语法的一部分。要想把双引号或反斜线放在被引用的界限值中，需要在双引号或反斜线前面加一个反斜线（在双引号引用的界限值中的一对双引号表示一个双引号字符，这与SQL字符串中的单引号规则类似）。此外，可以避免引用并且使用反斜线转义来保护所有数据字符，否则它们会被当做返回语法的一部分。什么都不写则表示一个无限界限，因此，要表示空字符串的界限值，可以写成""。

范围值前后允许有空格，但是圆括号或方括号之间的任何空格会被当做上下界值的一部分（取决于元素类型，它可能是有意义的也可能是无意义的）。

示例

查询包括3，不包括7，并且包括3和7之间的所有点：

```
SELECT '[3,7)'::int4range;
 int4range
```

```
-----------
 [3,7)
(1 row)
```

查询既不包括3也不包括 7，但是包括之间的所有点：

```
SELECT '(3,7)'::int4range;
 int4range
-----------
 [4,7)
(1 row)
```

查询只包括单独一个点4：

```
SELECT '[4,4]'::int4range;
 int4range
-----------
 [4,5)
(1 row)
```

查询不包括点（并且将被标准化为 '空'）：

```
SELECT '[4,4)'::int4range;
 int4range
-----------
 empty
(1 row)
```

构造范围

每一种范围类型都有一个与其同名的构造器函数。使用构造器函数比写一个范围文字常数更方便，因为它避免了对界限值的额外引用。构造器函数接受两个或三个参数。两个参数的形式以标准的形式构造一个范围（包含下界，排除上界），而三个参数的形式按照第三个参数指定的界限形式构造一个范围。第三个参数必须是下列字符串之一：“()”、“(]”、“[)”或者“[]”。例如：

完整形式是：下界、上界以及指示界限包含性/排除性的文本参数：

```
SELECT numrange(1.0, 14.0, '(]');
  numrange
------------
 (1.0,14.0]
(1 row)
```

如果第三个参数被忽略，则假定为 '[)'：

```
SELECT numrange(1.0, 14.0);
  numrange
------------
 [1.0,14.0)
(1 row)
```

尽管这里指定了'(]'，单返回结果时该值将被转换成标准形式，因为int8range是一种离散范围类型：

```
SELECT int8range(1, 14, '(]');
 int8range
-----------
 [2,15)
(1 row)
```

界限使用NULL导致范围是无界的：

```
SELECT numrange(NULL, 2.2);
 numrange
----------
 (,2.2)
(1 row)
```

离散范围类型

离散范围是指其元素类型具有定义明确的“步长”的范围，如integer或date。在这些类型中，当两个元素之间没有有效值时，它们可以被说成是相邻。

离散范围类型的每个元素值都有一个明确的“下一个”或“上一个”值。这样就可以通过选择下一个或上一个元素值，在范围界限的包含和排除表达之间转换。例如，在整数范围类型中，[4,8]和(3,9)表示相同的值集合，但对于超过numeric的范围，情况并非如此。

离散范围类型应具有识别元素类型所需步长的规范函数。规范化函数负责将范围类型的等价值转换为具有相同的表示，特别是与包含或者排除界限一致。如果未指定规范化函数，则具有不同格式的范围将始终被视为不相等，即使它们实际上是表达相同的一组值。

内置范围类型int4range、int8range和daterange都使用规范形式，该形式包括下界并且排除上界，也就是[)。但是，用户定义的范围类型可以使用其他约定。

自定义范围类型

用户可以定义范围类型。例如，要创建一个 subtype float8的范围类型：

```
CREATE TYPE floatrange AS RANGE (
        subtype = float8,
        subtype_diff = float8mi
    );

SELECT '[1.234, 5.678]'::floatrange;
```

索引

可以为范围类型的表列创建GiST索引。例如：

```
CREATE TABLE reservation (room int, during tsrange);
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

GiST索引可以加速涉及以下范围操作符的查询： =、 &&、 <@、 @>、 <<、 >>、 -|-、 &<以及 &> 。详见**范围操作符**。

此外，也可以在范围类型的表列上创建B-tree索引。对于这些索引类型，有用的范围操作就是等值。范围类型的B-tree支持主要是为了允许在查询内部进行排序，而不是创建真正的索引。

4.20 复合类型

复合类型表示行或记录的结构，它本质上就是字段名及其数据类型的列表。GaussDB(DWS)允许支持将表的列声明为复合类型。复合类型本质上和表的行类型相同，但是如果只想定义一种类型，使用**CREATE TYPE**可避免创建一个实际的表。单独的复合类型也是很有用的，例如可以作为函数的参数或者返回类型。

复合类型的声明

GaussDB(DWS)支持用户使用**CREATE TYPE**定义复合类型：

```
CREATE TYPE complex AS (
        r       double precision,
        i       double precision
    );
```

```
CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

定义复合类型之后，可用来创建表或函数：

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
    AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

构造复合值

要把复合值写作文字常量，可以将字段值括在圆括号中，并用逗号分隔。可以在任何字段值加上双引号，如果字段值包含逗号或括号则必须这样做。复合常量的一般格式如下：

```
'( val1 , val2 , ... )'
```

上文中的'("fuzzy dice",42,1.99)'便属于inventory_item类型的一个合法值。

要让一个字段为NULL，在列表中对应位置上空出即可。如果需要一个字段为空字符串，使用引号即可。例如下列示例，第一个字段是非NULL空字符串，第三个是NULL：

```
'("",42,)'
```

ROW表达式也能被用来构建组合值。例如：

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

访问组合类型

要访问复合列的字段，可以写成一个点和字段的名称，就像从表名中选择字段一样。为了避免混淆，必须使用括号进行区分。例如，尝试从示例表on_hand中选择一些子域：

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
ERROR:  missing FROM-clause entry for table "item"
```

这样写是会与从表中选择字段混淆的，因为名称item会被当成是一个表名，而不是on_hand的一个列名。必须写成这样：

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

或者还需要使用表名（例如在一个多表查询中）：

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

现在加上括号的对象就被正确地解释为对item列的引用，然后可以从中选出子域。

只要从一个组合值中选择一个字段，相似的语法问题就适用。例如，要从返回组合值的函数的结果中选取一个字段，需要这样写：

```
SELECT (my_func(...)).field FROM ...
```

如果没有额外的圆括号，这将生成一个语法错误。

4.21 列存表支持的数据类型

列存表支持的数据类型如表4-24所示。

表 4-24 列存表支持的数据类型

类别	数据类型	长度	是否支持
数值类型	smallint	2	支持
	integer	4	支持
	bigint	8	支持
	decimal	可变长度	支持
	numeric	可变长度	支持
	real	4	支持
	double precision	8	支持
	smallserial	2	支持
	serial	4	支持
	bigserial	8	支持
货币类型	money	8	支持
字符类型	character varying(n), varchar(n)	可变长度	支持
	character(n), char(n)	n	支持
	character、char	1	支持
	text	可变长度	支持
	nvarchar2	可变长度	支持
日期/时间类型	timestamp with time zone	8	支持
	timestamp without time zone	8	支持
	date	4	支持
	time without time zone	8	支持
	time with time zone	12	支持

类别	数据类型	长度	是否支持
	interval	16	支持
二进制类型	clob	可变长度	支持
	blob	可变长度	不支持
其他类型	…	…	不支持

4.22 XML 类型

XML数据类型可以被用来存储XML（eXtensible Markup Language）数据。XML数据可以存储为text类型，但是XML数据类型的优势在于会检查每个存储的值是不是结构良好的XML值。XML可以存储由XML标准定义的格式良好的文档，以及由XML标准中定义的“内容”片断，内容片断中可以有多个顶级元素或字符节点。

XML类型相关的支持函数请参见**XML函数**。

设置 XML 参数

语法格式如下：

```
SET XML OPTION { DOCUMENT | CONTENT };
SET xmloption TO { DOCUMENT | CONTENT };
```

当一个字符串值在没有通过XMLPARSE或XMLSERIALIZE函数与XML类型进行转换时，由XML OPTION会话配置参数来决定选择DOCUMENT还是CONTENT。

默认为CONTENT，表示所有形式的xml数据都被允许。

示例：

```
SET XML OPTION DOCUMENT;
SET
SET xmloption TO DOCUMENT;
SET
```

设置二进制数据的编码格式

语法格式：

```
SET xmlbinary TO { base64 | hex};
```

示例：

```
SET xmlbinary TO base64;
SET

SELECT xmlelement(name foo, bytea 'bar');
xmlelement
-----------------
<foo>YmFy</foo>
(1 row)

SET xmlbinary TO hex;
```

```
SET

SELECT xmlelement(name foo, bytea 'bar');
xmlelement
------------------
<foo>626172</foo>
(1 row)
```

访问 XML 值

XML数据类型比较特殊，它不提供任何比较操作符，这是因为对于XML数据没有通用的比较算法，所以无法通过比较一个XML值和一个搜索值来检索数据行。XML数据通常应该伴随一个ID值用于检索数据。另一种比较XML值的方案是将XML值转换成字符串，但字符串的比较并不能解决常见的XML值比较场景。

5 常量与宏

GaussDB(DWS)支持的常量和宏请参见表5-1。

表 5-1 常量和宏

参数	描述	相关SQL语句示例
CURRENT_CATALOG	当前数据库	SELECT CURRENT_CATALOG;
CURRENT_ROLE	当前角色	SELECT CURRENT_ROLE;
CURRENT_SCHEMA	当前数据库模式	SELECT CURRENT_SCHEMA;
CURRENT_USER	当前用户	SELECT CURRENT_USER;
LOCALTIMESTAMP	当前会话时间（无时区）	SELECT LOCALTIMESTAMP;
NULL	空值	-
SESSION_USER	当前系统用户	SELECT SESSION_USER;
SYSDATE	当前系统日期	SELECT SYSDATE; 或 SELECT now()::DATE;
USER	当前用户，与CURRENT_USER一致。	SELECT USER;

6 函数和操作符

6.1 字符处理函数和操作符

GaussDB(DWS)提供的字符处理函数和操作符主要用于字符串与字符串、字符串与非字符串之间的连接，以及字符串的模式匹配操作。

ascii(string)

描述：参数string的第一个字符的ASCII码。

返回值类型：integer

示例：

```
SELECT ascii('xyz');
 ascii
-------
   120
(1 row)
```

bit_length(string)

描述：字符串的位数。

返回值类型：integer

示例：

```
SELECT bit_length('world');
 bit_length
------------
         40
(1 row)
```

btrim(string text [, characters text])

描述：从string开头和结尾删除只包含characters中字符（缺省是空白）的最长字符串。

返回值类型：text

示例：


```
SELECT btrim('sring' , 'ing');
 btrim
-------
 sr
(1 row)
```

char_length(string)或 character_length(string)

描述：字符串中的字符个数。

返回值类型：integer

示例：

```
SELECT char_length('hello');
 char_length
-------------
           5
(1 row)
```

chr(integer)

描述：给出ASCII码的字符。

返回值类型：varchar

示例：

```
SELECT chr(65);
 chr
-----
 A
(1 row)
```

concat(str1,str2)

描述：将字符串str1和str2连接并返回。

- ORA和TD兼容模式下，返回结果为所有非NULL字符串的连接。
- MySQL兼容模式下，入参中存在NULL时，返回结果为NULL。

返回值类型：varchar

示例：

```
SELECT concat('Hello', ' World!');
    concat
--------------
 Hello World!
(1 row)
```

concat_ws(sep text, str"any" [, str"any" [, ...]])

描述：以第一个参数为分隔符，链接第二个以后的所有参数。

返回值类型：text

示例：

```
SELECT concat_ws(',', 'ABCDE', 2, NULL, 22);
 concat_ws
------------
 ABCDE,2,22
(1 row)
```


convert(string bytea, src_encoding name, dest_encoding name)

描述：以dest_encoding指定的目标编码方式转化字符串bytea。src_encoding指定源编码方式，在该编码下，string必须是合法的。

返回值类型：bytea

示例：

```
SELECT convert('text_in_utf8', 'UTF8', 'GBK');
         convert
---------------------------
 \x746578745f696e5f75746638
(1 row)
```

说明

如果源编码格式到目标编码格式的转化规则不存在，则字符串不进行任何转换直接返回，如GBK和LATIN1之间的转换规则是不存在的，具体转换规则可以通过查看系统表pg_conversion获得。

示例：

```
show server_encoding;
 server_encoding
-----------------
 LATIN1
(1 row)

SELECT convert_from('some text', 'GBK');
 convert_from
--------------
 some text
(1 row)

db_latin1=# SELECT convert_to('some text', 'GBK');
      convert_to
----------------------
 \x736f6d652074657874
(1 row)

db_latin1=# SELECT convert('some text', 'GBK', 'LATIN1');
       convert
----------------------
 \x736f6d652074657874
(1 row)
```

convert_from(string bytea, src_encoding name)

描述：以数据库的编码方式转化字符串bytea。

src_encoding指定源编码方式，在该编码下，string必须是合法的。

返回值类型：text

示例：

```
SELECT convert_from('text_in_utf8', 'UTF8');
 convert_from
--------------
 text_in_utf8
(1 row)
SELECT convert_from('\x6461746162617365','gbk');
 convert_from
--------------
 database
(1 row)
```

convert_to(string text, dest_encoding name)

描述：将字符串转化为dest_encoding的编码格式。

返回值类型：bytea

示例：

```
SELECT convert_to('some text', 'UTF8');
      convert_to
----------------------
 \x736f6d652074657874
(1 row)
SELECT convert_to('database', 'gbk');
     convert_to
--------------------
 \x6461746162617365
(1 row)
```

decode(string text, format text)

描述：将二进制数据从文本数据中解码。

返回值类型：bytea

示例：

```
SELECT decode('ZGF0YWJhc2U=', 'base64');
    decode
--------------
 \x6461746162617365
(1 row)

SELECT convert_from('\x6461746162617365','utf-8');
 convert_from
--------------
 database
(1 row)
```

encode(data bytea, format text)

描述：将二进制数据编码为文本数据。

返回值类型：text

示例：

```
SELECT encode('database', 'base64');
  encode
----------
 ZGF0YWJhc2U=
(1 row)
```

format(formatstr text [, str"any" [, ...]])

描述：格式化字符串。

返回值类型：text

示例：

```
SELECT format('Hello %s, %1$s', 'World');
       format
--------------------
```



```
 Hello World, World
(1 row)
```

instr(text,text,int,int)

描述：字符串匹配函数的位置，第一个int表示开始匹配起始位置，第二个int表示匹配的次数。

返回值类型：integer

示例：

```
SELECT instr( 'abcdabcdabcd', 'bcd', 2, 2 );
 instr
-------
     6
(1 row)
```

initcap(string)

描述：将字符串中的每个单词的首字母转化为大写，其他字母转化为小写。

返回值类型：text

示例：

```
SELECT initcap('hi THOMAS');
  initcap
-----------
 Hi Thomas
(1 row)
```

instr(string,substring[,position,occurrence])

描述：从字符串string的position（缺省时为1）所指的位置开始查找并返回第occurrence（缺省时为1）次出现子串substring的位置的值。

- 当position为0时，返回0。
- 当position为负数时，从字符串倒数第n个字符往前逆向搜索。n为position的绝对值。

本函数以字符为计算单位，如一个汉字为一个字符。

返回值类型：integer

示例：

```
SELECT instr('corporate floor','or', 3);
 instr
-------
     5
(1 row)
SELECT instr('corporate floor','or',-3,2);
 instr
-------
     2
(1 row)
```

lcase(string)

描述：把字符串转化为小写。

返回值类型：varchar

示例：

```
SELECT lcase('SAM');
 lcase
-------
 sam
(1 row)
```

left(str text, n int)

描述：返回字符串的前n个字符。

返回值类型：text

- ORA和TD兼容模式下，当n是负数时，返回除最后|n|个字符以外的所有字符。
- MySQL兼容模式下，当n是负数时，返回空串。

示例：

```
SELECT left('abcde', 2);
 left
------
 ab
(1 row)
```

length(string)

描述：获取参数string中字符的数目。

返回值类型：integer

示例：

```
SELECT length('abcd');
 length
--------
      4
(1 row)
```

length(string bytea, encoding name)

描述：指定encoding编码格式的string的字符数。在这个编码格式中，string必须是有效的。

返回值类型：integer

示例：

```
SELECT length('jose', 'UTF8');
 length
--------
      4
(1 row)
```

lengthb(string)

描述：获取参数string中字节的数目。与字符集有关，同样的中文字符，在GBK与UTF8中，返回的字节数不同。

返回值类型：integer

示例：


```
SELECT lengthb('hello');
 lengthb
---------
       5
(1 row)
```

lengthb(text/bpchar)

描述：获取指定字符串的字节数。

返回值类型：integer

示例：

```
SELECT lengthb('hello');
 lengthb
---------
       5
(1 row)
```

说明

- 若字符串中存在换行符，如字符串由一个换行符和一个空格组成，在GaussDB(DWS)中LENGTH和LENGTHB的值为2。
- 对于CHAR(n) 类型，GaussDB(DWS)中n是指字符个数。因此，对于多字节编码的字符集，LENGTHB函数返回的长度可能大于n。

locate(substring,string[,position])

描述：从字符串string的position（缺省时为1）所指的位置开始查找并返回第一次出现子串substring位置的值。以字符为计算单位。当string中不存在substring时，返回0。

返回值类型：integer

示例：

```
SELECT locate('ball','football');
 locate
--------
      5
(1 row)
SELECT locate('er','soccerplayer','6');
 locate
--------
     11
(1 row)
```

lower(string)

描述：把字符串转化为小写。

返回值类型：varchar

示例：

```
SELECT lower('TOM');
 lower
-------
 tom
(1 row)
```


lpad(string text, length int [, fill text])

描述：通过填充字符fill（缺省时为空白），把string填充为length长度。如果string已经比length长则将其尾部截断。

返回值类型：text

示例：

```
SELECT lpad('hi', 5, 'xyza');
 lpad
-------
 xyzhi
(1 row)
```

lpad(string varchar, length int[, repeat_string varchar])

描述：在string的左侧添上一系列的repeat_string（缺省为空白）来组成一个总长度为n的新字符串。

如果string本身的长度比指定的长度length长，则本函数将把string截断并把前面长度为length的字符串内容返回。

返回值类型：varchar

示例：

```
SELECT lpad('PAGE 1',15,'*.');
      lpad
-----------------
 *.*.*.*.*PAGE 1
(1 row)
SELECT lpad('hello world',5,'abcd');
 lpad
-------
 hello
(1 row)
```

octet_length(string)

描述：字符串中的字节数。

返回值类型：integer

示例：

```
SELECT octet_length('jose');
 octet_length
--------------
            4
(1 row)
```

overlay(string placing string FROM int [for int])

描述：替换子字符串。FROM int表示从第一个string的第几个字符开始替换，for int表示替换第一个string的字符数目。

返回值类型：text

示例：

```
SELECT overlay('hello' placing 'world' from 2 for 3 );
 overlay
```



```
---------
 hworldo
(1 row)
```

pg_client_encoding()

描述：当前客户端编码名称。

返回值类型：name

示例：

```
SELECT pg_client_encoding();
 pg_client_encoding
--------------------
 UTF8
(1 row)
```

position(substring in string)

描述：指定子字符串的位置。若string中没有substring，则返回0。

返回值类型：integer

示例：

```
SELECT position('ing' in 'string');
 position
----------
        4
(1 row)

SELECT position('ing' in 'strin');
 position
----------
        0
(1 row)
```

quote_ident(string text)

描述：返回适用于SQL语句的标识符形式（使用适当的引号进行界定）。只有在必要的时候才会添加引号（字符串包含非标识符字符或者会转换大小写的字符）。返回值中嵌入的引号会被双写。

返回值类型：text

示例：

```
SELECT quote_ident('hello world');
 quote_ident
--------------
 "hello world"
(1 row)
```

quote_literal(string text)

描述：返回适用于在SQL语句里当作文本使用的形式（使用适当的引号进行界定）。

返回值类型：text

示例：

```
SELECT quote_literal('hello');
 quote_literal
```



```
---------------
 'hello'
(1 row)
```

如果出现如下写法，text文本将进行转义。

```
SELECT quote_literal(E'O\'hello');
 quote_literal
---------------
 'O''hello'
(1 row)
```

如果出现如下写法，反斜杠会写入两次。

```
SELECT quote_literal('O\hello');
 quote_literal
---------------
 E'O\\hello'
(1 row)
```

如果参数为NULL，返回空。如果参数可能为null，通常使用函数quote_nullable更适用。

```
SELECT quote_literal(NULL);
 quote_literal
---------------

(1 row)
```

quote_literal(value anyelement)

描述：将给定的值强制转换为text，加上引号作为文本。

返回值类型：text

示例：

```
SELECT quote_literal(42.5);
 quote_literal
---------------
 '42.5'
(1 row)
```

如果出现如下写法，定值将进行转义。

```
SELECT quote_literal(E'O\'42.5');
 quote_literal
---------------
 '0''42.5'
(1 row)
```

如果出现如下写法，反斜杠会写入两次。

```
SELECT quote_literal('O\42.5');
 quote_literal
---------------
 E'O\\42.5'
(1 row)
```

quote_nullable(string text)

描述：返回适用于在SQL语句里当作字符串使用的形式（使用适当的引号进行界定）。

返回值类型：text

示例：


```
SELECT quote_nullable('hello');
 quote_nullable
----------------
 'hello'
(1 row)
```

如果出现如下写法，text文本将进行转义。

```
SELECT quote_nullable(E'O\'hello');
 quote_nullable
----------------
 'O''hello'
(1 row)
```

如果出现如下写法，反斜杠会写入两次。

```
SELECT quote_nullable('O\hello');
 quote_nullable
----------------
 E'O\\hello'
(1 row)
```

如果参数为NULL，返回NULL。

```
SELECT quote_nullable(NULL);
 quote_nullable
----------------
 NULL
(1 row)
```

quote_nullable(value anyelement)

描述：将给定的参数值转化为text，加上引号作为文本。

返回值类型：text

示例：

```
SELECT quote_nullable(42.5);
 quote_nullable
----------------
 '42.5'
(1 row)
```

如果出现如下写法，定值将进行转义。

```
SELECT quote_nullable(E'O\'42.5');
 quote_nullable
----------------
 'O''42.5'
(1 row)
```

如果出现如下写法，反斜杠会写入两次。

```
SELECT quote_nullable('O\42.5');
 quote_nullable
----------------
 E'O\\42.5'
(1 row)
```

如果参数为NULL，返回NULL。

```
SELECT quote_nullable(NULL);
 quote_nullable
----------------
 NULL
(1 row)
```


rawcat(raw,raw)

描述：字符串拼接函数。

返回值类型：raw

示例：

```
SELECT rawcat('ab','cd');
 rawcat
--------
 ABCD
(1 row)
```

regexp_like(source_string, pattern [, match_parameter])

描述：正则表达式的模式匹配函数。

source_string为源字符串，pattern为正则表达式匹配模式。 match_parameter为匹配选项，可取值为：

- 'i'：大小写不敏感。
- 'c'：大小写敏感。
- 'n'：允许正则表达式元字符“.”匹配换行符。
- 'm'：将source_string视为多行。

若忽略match_parameter选项，默认为大小写敏感，“.”不匹配换行符，source_string视为单行。

返回值类型：boolean

示例：

```
SELECT regexp_like('ABC', '[A-Z]');
 regexp_like
-------------
 t
(1 row)
SELECT regexp_like('ABC', '[D-Z]');
 regexp_like
-------------
 f
(1 row)
SELECT regexp_like('abc', '[A-Z]','i');
 regexp_like
-------------
 t
(1 row)
SELECT regexp_like('abc', '[A-Z]');
 regexp_like
-------------
 f
(1 row)
```

regexp_like(text,text,text)

描述：正则表达式的模式匹配函数。

返回值类型：bool

示例：

```
SELECT regexp_like('str','[ac]');
 regexp_like
```



```
-------------
 f
(1 row)
```

regexp_matches(string text, pattern text [, flags text])

描述：返回string中所有匹配POSIX正则表达式的子字符串。如果pattern不匹配，该函数不返回行。如果模式不包含圆括号子表达式，则每一个被返回的行都是一个单一元素的文本数组，其中包括匹配整个模式的子串。如果模式包含圆括号子表达式，该函数返回一个文本数组，它的第n个元素是匹配模式的第n个圆括号子表达式的子串。

flags参数为可选参数，包含零个或多个改变函数行为的单字母标记。i表示进行大小写无关的匹配，g表示替换每一个匹配的子字符串而不仅仅是第一个。

须知

如果提供了最后一个参数，但参数值是空字符串（''），且数据库SQL兼容模式设置为ORA的情况下，会导致返回结果为空集。这是因为ORA兼容模式将''作为NULL处理，避免此类行为的方式有如下几种：

- 将数据库SQL兼容模式改为TD；
- 不提供最后一个参数，或最后一个参数不为空字符串。

返回值类型：setof text[]

示例：

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
 regexp_matches
----------------
 {bar,beque}
(1 row)
SELECT regexp_matches('foobarbequebaz', 'barbeque');
 regexp_matches
----------------
 {barbeque}
(1 row)
 SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
    result
---------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

regexp_replace(string, pattern, replacement [,flags])

描述：替换匹配POSIX正则表达式的子字符串。 如果没有匹配pattern，那么返回不加修改的string串。 如果有匹配，则返回的string串里面的匹配子串将被replacement串替换掉。

replacement串可以包含\n， 其中\n是1到9， 表明string串里匹配模式里第n个圆括号子表达式的子串应该被插入， 并且它可以包含\&表示应该插入匹配整个模式的子串。

可选的flags参数包含零个或多个改变函数行为的单字母标记，见下表。

表 6-1 flags 参数的可选项

选项	描述
g	表示替换每一个匹配的子字符串而不仅仅是第一个（默认仅替换第一个匹配的子字符串）。
B	默认情况下使用Henry Spencer正则库及其正则语法。指定B选项后，表示优先选用boost regex正则库及其正则语法。 以下两种情况在指定了B选项时，也会自动转换为选择Henry Spencer正则库及其正则语法： ● flags同时指定了p、q、w、x中的任意个字符。 ● string或pattern参数中含有多字节字符。
b	表示按照BRE(POSIX Basic Regular Expression)匹配模式的规则进行匹配。
c	大小写敏感匹配。
e	表示按照ERE(POSIX Extended Regular Expression)匹配模式的规则进行匹配。当b和e都未指定时，如果选用的是Henry Spencer正则库，则按照ARE(Advanced Regular Expression，类似于Perl Regular Expression)匹配模式的规则进行匹配；如果选用的是boost regex正则库，则按照Perl Regular Expression匹配模式的规则进行匹配。
i	大小写不敏感匹配。
m	换行敏感匹配，与选项n同义。
n	换行敏感匹配。此选项生效时，换行符影响元字符(.、^、$和[^)的匹配。
p	部分换行敏感匹配，此选项生效时，换行符影响元字符(.和[^)的匹配。部分是相对选项n而言。
q	重置正则表达式为加双引号的文本字符串，所有都是普通字符。
s	非换行敏感匹配。
t	紧凑语法（缺省）。该选项生效时，所有字符都很重要。
w	反部分换行敏感匹配，此选项生效时，换行符影响元字符(^和$)的匹配。部分是相对选项n而言。
x	扩展语法。与紧凑语法相对，在扩展的语法中，正则表达式中的空白字符被忽略。空白字符包括空格、水平制表符、新行、和任何属于space字符表的字符。

返回值类型：varchar

示例：

```
SELECT regexp_replace('Thomas', '.[mN]a.', 'M');
 regexp_replace
----------------
 ThM
(1 row)
SELECT regexp_replace('foobarbaz','b(..)', E'X\\1Y', 'g') AS RESULT;
```



```
  result
-------------
 fooXarYXazY
(1 row)
```

regexp_substr(text,text)

描述：正则表达式的抽取子串函数。与substr功能相似，正则表达式出现多个并列的括号时，也全部处理。

返回值类型：text

示例：

```
SELECT regexp_substr('str','[ac]');
 regexp_substr
---------------

(1 row)
```

regexp_split_to_array(string text, pattern text [, flags text])

描述：用POSIX正则表达式作为分隔符，分隔string。和regexp_split_to_table相同，不过regexp_split_to_array会把它的结果以一个text数组的形式返回。

返回值类型：text[]

示例：

```
SELECT regexp_split_to_array('hello world', E'\\s+');
 regexp_split_to_array
-----------------------
 {hello,world}
(1 row)
```

regexp_split_to_table(string text, pattern text [, flags text])

描述：用POSIX正则表达式作为分隔符，分隔string。如果没有与pattern的匹配，该函数返回string。如果有至少有一个匹配，对每一个匹配它都返回从上一个匹配的末尾（或者串的开头）到这次匹配开头之间的文本。当没有更多匹配时，它返回从上一次匹配的末尾到串末尾之间的文本。

flags参数包含零个或多个改变函数行为的单字母标记。i表示进行大小写无关的匹配，g表示替换每一个匹配的子字符串而不仅仅是第一个。

返回值类型：setof text

示例：

```
SELECT regexp_split_to_table('hello world', E'\\s+');
 regexp_split_to_table
-----------------------
 hello
 world
(2 rows)
```

regexp_substr(source_char, pattern)

描述：正则表达式的抽取子串函数。

返回值类型：varchar

示例：


```
SELECT regexp_substr('500 Hello World, Redwood Shores, CA', ',[^,]+,') "REGEXPR_SUBSTR";
  REGEXPR_SUBSTR
------------------
 , Redwood Shores,
(1 row)
```

repeat(string text, number int)

描述：text

返回值类型：将string重复number次。

示例：

```
SELECT repeat('Pg', 4);
  repeat
----------
 PgPgPgPg
(1 row)
```

replace(string text, from text, to text)

描述：把字符串string里出现地所有子字符串from的内容替换成子字符串to的内容。

返回值类型：text

示例：

```
SELECT replace('abcdefabcdef', 'cd', 'XXX');
    replace
----------------
 abXXXefabXXXef
(1 row)
```

replace(string varchar, search_string varchar, replacement_string varchar)

描述：把字符串string中所有子字符串search_string替换成子字符串replacement_string。

返回值类型：varchar

示例：

```
SELECT replace('jack and jue','j','bl');
    replace
----------------
 black and blue
(1 row)
```

reverse(str)

描述：返回颠倒的字符串。

返回值类型：text

示例：

```
SELECT reverse('abcde');
 reverse
---------
 edcba
(1 row)
```


right(str text, n int)

描述：返回字符串中的后n个字符。

- ORA和TD兼容模式下，当n是负数时，返回除前|n|个字符以外的所有字符。
- MySQL兼容模式下，当n是负数时，返回空串。

返回值类型：text

示例：

```
SELECT right('abcde', 2);
 right
-------
 de
(1 row)

SELECT right('abcde', -2);
 right
-------
 cde
(1 row)
```

rpad(string varchar, length int [, fill varchar])

描述：使用填充字符fill（缺省时为空白），把string填充到length长度。如果string已经比length长则将其从尾部截断。

length参数在GaussDB(DWS)中表示字符长度。一个汉字长度计算为一个字符。

返回值类型：varchar

示例：

```
SELECT rpad('hi',5,'xyza');
 rpad
-------
 hixyz
(1 row)
SELECT rpad('hi',5,'abcdefg');
 rpad
-------
 hiabc
(1 row)
```

rpad(string text, length int [, fill text])

描述：使用填充字符fill（缺省时为空白），把string填充到length长度。如果string已经比length长则将其从尾部截断。

返回值类型：text

示例：

```
SELECT rpad('hi', 5, 'xy');
 rpad
-------
 hixyx
(1 row)
```

rtrim(string text [, characters text])

描述：从字符串string的结尾删除只包含characters中字符（缺省是个空白）的最长的字符串。

返回值类型：text

示例：

```
SELECT rtrim('trimxxxx', 'x');
 rtrim
-------
 trim
(1 row)
```

rtrim(string [, characters])

描述：从字符串string的结尾删除只包含characters中字符（缺省是个空白）的最长的字符串。

返回值类型：varchar

示例：

```
SELECT rtrim('TRIMxxxx','x');
 rtrim
-------
 TRIM
(1 row)
```

ltrim(string [, characters])

描述：从字符串string的开头删除只包含characters中字符（缺省是一个空白）的最长的字符串。

返回值类型：varchar

示例：

```
SELECT ltrim('xxxxTRIM','x');
 ltrim
-------
 TRIM
(1 row)
```

string || string

描述：连接字符串。

返回值类型：text

示例：

```
SELECT 'DA'||'TABASE' AS RESULT;
 result
--------
 DATABASE
(1 row)
```

string || non-string 或 non-string || string

描述：连接字符串和非字符串。

返回值类型：text

示例：

```
SELECT 'Value: '||42 AS RESULT;
  result
```



```
-----------
 Value: 42
(1 row)
```

substring(string [from int] [for int])

描述：截取子字符串，from int表示从第几个字符开始截取，for int表示截取几个字节。

返回值类型：text

示例：

```
SELECT substring('Thomas' from 2 for 3);
 substring
-----------
 hom
(1 row)
```

substring(string from *pattern*)

描述：截取匹配POSIX正则表达式的子字符串。如果没有匹配它返回空值，否则返回文本中匹配模式的那部分。

返回值类型：text

示例：

```
SELECT substring('Thomas' from '...$');
 substring
-----------
 mas
(1 row)
SELECT substring('foobar' from 'o(.)b');
 result
--------
 o
(1 row)
SELECT substring('foobar' from '(o(.)b)');
 result
--------
 oob
(1 row)
```

说明

如果POSIX正则表达式模式包含任何圆括号，那么将返回匹配第一对子表达式（对应第一个左圆括号的）的文本。如果想在表达式里使用圆括号而又不想导致这个例外，那么可以在整个表达式外边加上一对圆括号。

substring(string from *pattern* for *escape*)

描述：截取匹配SQL正则表达式的子字符串。声明的模式必须匹配整个数据串，否则函数失败并返回空值。为了标识在成功的时候应该返回的模式部分，模式必须包含逃逸字符的两次出现，并且后面要跟上双引号（"）。匹配这两个标记之间的模式的文本将被返回。

返回值类型：text

示例：

```
SELECT substring('Thomas' from '%#"o_a#"_' for '#');
 substring
```



```
-----------
 oma
(1 row)
```

split_part(string text, delimiter text, field int)

描述：根据delimiter分隔string返回生成的第field个子字符串（从出现第一个delimiter的text为基础）。

返回值类型：text

示例：

```
SELECT split_part('abc~@~def~@~ghi', '~@~', 2);
 split_part
------------
 def
(1 row)
```

strpos(string, substring)

描述：指定的子字符串的位置。和position(substring in string)一样，不过参数顺序相反。

返回值类型：integer

示例：

```
SELECT strpos('source', 'rc');
 strpos
--------
      4
(1 row)
```

substrb(text,int,int)

描述：提取子字符串，第一个int表示提取的起始位置，第二个表示提取几个字节。

返回值类型：text

示例：

```
SELECT substrb('string',2,3);
 substrb
---------
 tri
(1 row)
```

substrb(text,int)

描述：提取子字符串，int表示提取的起始位置。

返回值类型：text

示例：

```
SELECT substrb('string',2);
 substrb
---------
 tring
(1 row)
```


sys_context ('namespace' , 'parameter')

描述：获取并返回指定namespace下参数parameter的值。

返回值类型：text

示例：

```
SELECT SYS_CONTEXT ( 'postgres' , 'archive_mode');
 sys_context
-------------

(1 row)
```

to_hex(number int or bigint)

描述：把number转换成十六进制表现形式。

返回值类型：text

示例：

```
SELECT to_hex(2147483647);
  to_hex
----------
 7fffffff
(1 row)
```

translate(string text, from text, to text)

描述：把在string中包含的任何匹配from中字符的字符转化为对应的在to中的字符。如果from比to长，删掉在from中出现的额外的字符。

返回值类型：text

示例：

```
SELECT translate('12345', '143', 'ax');
 translate
-----------
 a2x5
(1 row)
```

substr(string,from)

描述：

从参数string中抽取子字符串。

from表示抽取的起始位置。

- from为0时，按1处理。
- from为正数时，抽取从from到末尾的所有字符。
- from为负数时，抽取字符串的后n个字符，n为from的绝对值。

返回值类型：varchar

示例：

from为正数时：

```
SELECT substr('ABCDEF',2);
 substr
```



```
--------
 BCDEF
(1 row)
```

from为负数时：

```
SELECT substr('ABCDEF',-2);
 substr
--------
 EF
(1 row)
```

substr(string,from,count)

描述：

从参数string中抽取子字符串。

from表示抽取的起始位置。

count表示抽取的子字符串长度。

- from为0时，按1处理。
- from为正数时，抽取从from开始的count个字符。
- from为负数时，抽取从倒数第n个开始的count个字符，n为from的绝对值。
- count小于1时，返回null。

返回值类型：varchar

示例：

from为正数时：

```
SELECT substr('ABCDEF',2,2);
 substr
--------
 BC
(1 row)
```

from为负数时：

```
SELECT substr('ABCDEF',-3,2);
 substr
--------
 DE
(1 row)
```

substrb(string,from)

描述：该函数和SUBSTR(string,from)函数功能一致，但是计算单位为字节。

返回值类型：bytea

示例：

```
SELECT substrb('ABCDEF',-2);
 substrb
---------
 EF
(1 row)
```


substrb(string,from,count)

描述：该函数和SUBSTR(string,from,count)函数功能一致，但是计算单位为字节。

返回值类型：bytea

示例：

```
SELECT substrb('ABCDEF',2,2);
 substrb
---------
 BC
(1 row)
```

string [NOT] LIKE pattern [ESCAPE escape-character]

描述：模式匹配函数。

如果pattern不包含百分号或者下划线，该模式只代表它本身，这时候LIKE的行为就像等号操作符。在pattern里的下划线（_）匹配任何单个字符；而一个百分号（%）匹配零或多个任何字符。

要匹配下划线或者百分号本身，在pattern里相应的字符必须前导逃逸字符。缺省的逃逸字符是反斜杠，但是用户可以用ESCAPE子句指定一个。要匹配逃逸字符本身，写两个逃逸字符。

返回值类型：boolean

示例：

```
SELECT 'AA_BBCC' LIKE '%A@_B%' ESCAPE '@' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'AA_BBCC' LIKE '%A@_B%' AS RESULT;
 result
--------
 f
(1 row)
SELECT 'AA@_BBCC' LIKE '%A@_B%' AS RESULT;
 result
--------
 t
(1 row)
```

trim([leading |trailing |both] [characters] from string)

描述：从字符串string的开头、结尾或两边删除只包含characters中字符（缺省是一个空白）的最长的字符串。

返回值类型：varchar

示例：

```
SELECT trim(BOTH 'x' FROM 'xTomxx');
 btrim
-------
 Tom
(1 row)
SELECT trim(LEADING 'x' FROM 'xTomxx');
 ltrim
-------
 Tomxx
(1 row)
```



```
SELECT trim(TRAILING 'x' FROM 'xTomxx');
 rtrim
-------
 xTom
(1 row)
```

ucase(string)

描述：把字符串转化为大写。

返回值类型：varchar

示例：

```
SELECT ucase('sam');
 ucase
-------
 SAM
(1 row)
```

upper(string)

描述：把字符串转化为大写。

返回值类型：varchar

示例：

```
SELECT upper('tom');
 upper
-------
 TOM
(1 row)
```

6.2 二进制字符串函数和操作符

SQL定义了一些二进制字符串函数，这些函数使用关键字而不是逗号来分隔参数。另外，DWS提供了函数调用所使用的常用语法。

octet_length(string)

描述：二进制字符串中的字节数。

返回值类型：integer

示例：

```
SELECT octet_length(E'jo\\000se'::bytea) AS RESULT;
 result
--------
      5
(1 row)
```

overlay(string placing string from int [for int])

描述：替换子串。

返回值类型：bytea

示例：


```
SELECT overlay(E'Th\\000omas'::bytea placing E'\\002\\003'::bytea from 2 for 3) AS RESULT;
    result
----------------
 \x5402036d6173
(1 row)
```

position(substring in string)

描述：特定子字符串的位置。

返回值类型：integer

示例：

```
SELECT position(E'\\000om'::bytea in E'Th\\000omas'::bytea) AS RESULT;
 result
--------
      3
(1 row)
```

substring(string [from int] [for int])

描述：截取子串。

返回值类型：bytea

示例：

```
SELECT substring(E'Th\\000omas'::bytea from 2 for 3) AS RESULT;
  result
----------
 \x68006f
(1 row)
```

截取时间，获取小时数：

```
select substring('2022-07-18 24:38:15',12,2)AS RESULT;
 result
-----------
 24
(1 row)
```

trim([both] bytes from string)

描述：从string的开头和结尾删除只包含bytes中字节的最长字符串。

返回值类型：bytea

示例：

```
SELECT trim(E'\\000'::bytea from E'\\000Tom\\000'::bytea) AS RESULT;
  result
----------
 \x546f6d
(1 row)
```

btrim(string bytea,bytes bytea)

描述：从string的开头和结尾删除只包含bytes中字节的最长的字符串。

返回值类型：bytea

示例：


```
SELECT btrim(E'\\000trim\\000'::bytea, E'\\000'::bytea) AS RESULT;
  result
------------
 \x7472696d
(1 row)
```

get_bit(string, offset)

描述：从字符串中抽取位。

返回值类型：integer

示例：

```
SELECT get_bit(E'Th\\000omas'::bytea, 45) AS RESULT;
 result
--------
      1
(1 row)
```

get_byte(string, offset)

描述：从字符串中抽取字节。

返回值类型：integer

示例：

```
SELECT get_byte(E'Th\\000omas'::bytea, 4) AS RESULT;
 result
--------
    109
(1 row)
```

set_bit(string,offset, newvalue)

描述：设置字符串中的位。

返回值类型：bytea

示例：

```
SELECT set_bit(E'Th\\000omas'::bytea, 45, 0) AS RESULT;
      result
------------------
 \x5468006f6d4173
(1 row)
```

set_byte(string,offset, newvalue)

描述：设置字符串中的字节。

返回值类型：bytea

示例：

```
SELECT set_byte(E'Th\\000omas'::bytea, 4, 64) AS RESULT;
      result
------------------
 \x5468006f406173
(1 row)
```


6.3 位串函数和操作符

除了常用的比较操作符之外，还可以使用以下的操作符。&，|和#的位串操作数必须等长。在位移的时候，保留原始的位串长度（并以0填充）。

||

描述：位串之间进行连接。

示例：

```
SELECT B'10001' || B'011' AS RESULT;
  result
----------
 10001011
(1 row)
```

&

描述：位串之间进行“与”操作。

示例：

```
SELECT B'10001' & B'01101' AS RESULT;
 result
--------
 00001
(1 row)
```

|

描述：位串之间进行“或”操作。

示例：

```
SELECT B'10001' | B'01101' AS RESULT;
 result
--------
 11101
(1 row)
```

#

描述：位串之间如果不一致进行“或”操作。如果两个位串中对应位置都为1或者0，则该位置返回为0。

示例：

```
SELECT B'10001' # B'01101' AS RESULT;
 result
--------
 11100
(1 row)
```

~

描述：位串之间进行“非”操作。

示例：


```
SELECT ~B'10001'AS RESULT;
 result
----------
 01110
(1 row)
```

<<

描述：位串进行左移操作。

示例：

```
SELECT B'10001' << 3 AS RESULT;
 result
----------
 01000
(1 row)
```

>>

描述：位串进行右移操作。

示例：

```
SELECT B'10001' >> 2 AS RESULT;
 result
----------
 00100
(1 row)
```

下列SQL标准函数除了可以用于字符串之外，也可以用于位串：length，bit_length，octet_length，position，substring，overlay。

下列的函数用于位串和二进制字符串：get_bit，set_bit。当用于位串时，函数位数从字符串的第一位（最左边）作为0位。

另外，在整数和bit之间可以相互转换。示例：

```
SELECT 44::bit(10) AS RESULT;
   result
------------
 0000101100
(1 row)

SELECT 44::bit(3) AS RESULT;
 result
--------
 100
(1 row)

SELECT cast(-44 as bit(12)) AS RESULT;
    result
--------------
 111111010100
(1 row)

SELECT '1110'::bit(4)::integer AS RESULT;
 result
--------
     14
(1 row)
```

说明

只是转换为“bit”的意思是转换成bit(1)，因此只会转换成整数的最低位。

6.4 数字操作函数和操作符

6.4.1 数字操作符

+

描述：加

示例：

```
SELECT 2+3 AS RESULT;
 result
--------
      5
(1 row)
```

-

描述：减

示例：

```
SELECT 2-3 AS RESULT;
 result
--------
     -1
(1 row)
```

*****

描述：乘

示例：

```
SELECT 2*3 AS RESULT;
 result
--------
      6
(1 row)
```

/

描述：除（除法操作符不会取整）

示例：

```
SELECT 4/2 AS RESULT;
 result
--------
      2
(1 row)
SELECT 4/3 AS RESULT;
       result
------------------
 1.33333333333333
(1 row)
```


+/-

描述：正/负

示例：

```
SELECT -2 AS RESULT;
 result
--------
     -2
(1 row)
```

%

描述：模（求余）

示例：

```
SELECT 5%4 AS RESULT;
 result
--------
      1
(1 row)
```

@

描述：绝对值

示例：

```
SELECT @ -5.0 AS RESULT;
 result
--------
    5.0
(1 row)
```

^

描述：幂（指数运算）

MySQL兼容模式下，作用为异或，参见**位串函数和操作符**章节的操作符“#”。

示例：

```
SELECT 2.0^3.0 AS RESULT;
       result
--------------------
 8.0000000000000000
(1 row)
```

|/

描述：平方根

示例：

```
SELECT |/ 25.0 AS RESULT;
 result
--------
      5
(1 row)
```

||/

描述：立方根

示例：

```
SELECT ||/ 27.0 AS RESULT;
 result
--------
      3
(1 row)
```

!

描述：阶乘

示例：

```
SELECT 5! AS RESULT;
 result
--------
    120
(1 row)
```

!!

描述：阶乘（前缀操作符）

示例：

```
SELECT !!5 AS RESULT;
 result
--------
    120
(1 row)
```

&

描述：二进制AND

示例：

```
SELECT 91&15  AS RESULT;
 result
--------
     11
(1 row)
```

|

描述：二进制OR

示例：

```
SELECT 32|3  AS RESULT;
 result
--------
     35
(1 row)
```

#

描述：二进制XOR

示例：

```
SELECT 17#5  AS RESULT;
 result
--------
     20
(1 row)
```

~

描述：二进制NOT

示例：

```
SELECT ~1 AS RESULT;
 result
--------
     -2
(1 row)
```

<<

描述：二进制左移

示例：

```
SELECT 1<<4 AS RESULT;
 result
--------
     16
(1 row)
```

>>

描述：二进制右移

示例：

```
SELECT 8>>2 AS RESULT;
 result
--------
      2
(1 row)
```

6.4.2 数字操作函数

abs(x)

描述：绝对值。

返回值类型：和输入相同。

示例：

```
SELECT abs(-17.4);
 abs
------
 17.4
(1 row)
```

acos(x)

描述：反余弦。

返回值类型：double precision

示例：

```
SELECT acos(-1);
       acos
------------------
 3.14159265358979
(1 row)
```

asin(x)

描述：反正弦。

返回值类型：double precision

示例：

```
SELECT asin(0.5);
       asin
------------------
 .523598775598299
(1 row)
```

atan(x)

描述：反正切。

返回值类型：double precision

示例：

```
SELECT atan(1);
       atan
------------------
 .785398163397448
(1 row)
```

atan2(y, x)

描述：y/x的反正切。

返回值类型：double precision

示例：

```
SELECT atan2(2, 1);
      atan2
------------------
 1.10714871779409
(1 row)
```

bitand(integer, integer)

描述：计算两个数字与运算(&)的结果。

返回值类型：bigint

示例：

```
SELECT bitand(127, 63);
 bitand
--------
```



```
    63
(1 row)
```

cbrt(double precision)

描述：立方根。

返回值类型：double precision

示例：

```
SELECT cbrt(27.0);
 cbrt
------
    3
(1 row)
```

ceil(double precision or numeric)

描述：不小于参数的最小的整数。

返回值类型：与输入相同。

示例：

```
SELECT ceil(-42.8);
 ceil
------
  -42
(1 row)
```

ceiling(double precision or numeric)

描述：不小于参数的最小整数（ceil的别名）。

返回值类型：与输入相同。

示例：

```
SELECT ceiling(-95.3);
 ceiling
---------
     -95
(1 row)
```

cos(x)

描述：余弦。

返回值类型：double precision

示例：

```
SELECT cos(-3.1415927);
        cos
-------------------
 -.999999999999999
(1 row)
```

cot(x)

描述：余切。

返回值类型：double precision

示例：

```
SELECT cot(1);
       cot
------------------
 .642092615934331
(1 row)
```

degrees(double precision)

描述：把弧度转为角度。

返回值类型：double precision

示例：

```
SELECT degrees(0.5);
     degrees
------------------
 28.6478897565412
(1 row)
```

div(y numeric, x numeric)

描述：y除以x的商的整数部分。

返回值类型：numeric

示例：

```
SELECT div(9,4);
 div
-----
   2
(1 row)
```

exp(double precision or numeric)

描述：自然指数。

返回值类型：与输入相同。

示例：

```
SELECT exp(1.0);
        exp
--------------------
 2.7182818284590452
(1 row)
```

floor(double precision or numeric)

描述：不大于参数的最大整数。

返回值类型：与输入相同。

示例：

```
SELECT floor(-42.8);
 floor
-------
```



```
  -43
(1 row)
```

radians(double precision)

描述：把角度转为弧度。

返回值类型：double precision

示例：

```
SELECT radians(45.0);
     radians
------------------
 .785398163397448
(1 row)
```

random()

描述：0.0到1.0之间的随机数。

返回值类型：double precision

示例：

```
SELECT random();
      random
------------------
 .824823560658842
(1 row)
```

ln(double precision or numeric)

描述：自然对数。

返回值类型：与输入相同。

示例：

```
SELECT ln(2.0);
         ln
--------------------
 .6931471805599453
(1 row)
```

log(double precision or numeric)

描述：以10为底的对数。

- ORA和TD兼容模式下，表现为以10为底的对数。
- MySQL兼容模式下，表现为自然对数。

返回值类型：与输入相同。

示例：

```
-- ORA兼容模式
SELECT log(100.0);
         log
--------------------
 2.0000000000000000
(1 row)
-- TD兼容模式
```



```
SELECT log(100.0);
        log
--------------------
 2.0000000000000000
(1 row)
-- MySQL兼容模式
SELECT log(100.0);
        log
--------------------
 4.6051701859880914
(1 row)
```

log(b numeric, x numeric)

描述：以b为底的对数。

返回值类型：numeric

示例：

```
SELECT log(2.0, 64.0);
        log
--------------------
 6.0000000000000000
(1 row)
```

mod(x,y)

描述：x/y的余数（模）。如果x是0，则返回0。如果y是0，则返回x。

返回值类型：与参数类型相同。

示例：

```
SELECT mod(9,4);
 mod
-----
   1
(1 row)
SELECT mod(9,0);
 mod
-----
   9
(1 row)
```

pi()

描述：“π”常量。

返回值类型：double precision

示例：

```
SELECT pi();
        pi
-------------------
 3.14159265358979
(1 row)
```

power(a double precision, b double precision)

描述：a的b次幂。

返回值类型：double precision

示例：

```
SELECT power(9.0, 3.0);
       power
---------------------
 729.0000000000000000
(1 row)
```

round(double precision or numeric)

描述：离输入参数最近的整数。

返回值类型：与输入相同。

示例：

```
SELECT round(42.4);
 round
-------
    42
(1 row)

SELECT round(42.6);
 round
-------
    43
(1 row)
```

说明

当调用round函数时，数值类型将向零舍入，而（在大多数计算机上）实数和双精度型则以最接近的偶数为结果。

round(v numeric, s int)

描述：保留小数点后s位，s后一位进行四舍五入。

返回值类型：numeric

示例：

```
SELECT round(42.4382, 2);
 round
-------
 42.44
(1 row)
```

setseed(double precision)

描述：为随后的random()调用设置种子(-1.0到1.0之间，包含边界值)。

返回值类型：void

示例：

```
SELECT setseed(0.54823);
 setseed
---------

(1 row)
```

sign(double precision or numeric)

描述：输出此参数的符号。

返回值类型：-1表示负数，0表示0，1表示正数。

示例：

```
SELECT sign(-8.4);
 sign
------
   -1
(1 row)
```

sin(x)

描述：正弦。

返回值类型：double precision

示例：

```
SELECT sin(1.57079);
       sin
------------------
 .999999999979986
(1 row)
```

sqrt(x)

描述：平方根。

返回值类型：与输入相同。

示例：

```
SELECT sqrt(2.0);
       sqrt
-------------------
 1.414213562373095
(1 row)
```

tan(x)

描述：正切。

返回值类型：double precision

示例：

```
SELECT tan(20);
       tan
------------------
 2.23716094422474
(1 row)
```

trunc(double precision or numeric)

描述：截断（取整数部分）。

返回值类型：与输入相同。

示例：

```
SELECT trunc(42.8);
 trunc
-------
```



```
   42
(1 row)
```

trunc(v numeric, s int)

描述：截断为s位小数。

返回值类型：numeric

示例：

```
SELECT trunc(42.4382, 2);
 trunc
-------
 42.43
(1 row)
```

width_bucket(operand numeric, b1 numeric, b2 numeric, count int)

描述：设定分组范围的最小值、最大值和分组个数，构建指定个数的大小相同的分组，返回指定字段值落入的分组编号。b1为分组范围的最小值，b2为分组范围的最大值，count为分组的个数。

返回值类型：integer

示例：

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);
 width_bucket
--------------
            3
(1 row)
```

width_bucket(op double precision, b1 double precision, b2 double precision, count int)

描述：设定分组范围的最小值、最大值和分组个数，构建指定个数的大小相同的分组，返回指定字段值落入的分组编号。b1为分组范围的最小值，b2为分组范围的最大值，count为分组的个数。

返回值类型：integer

示例：

```
SELECT width_bucket(5.35, 0.024, 10.06, 5);
 width_bucket
--------------
            3
(1 row)
```

6.5 时间、日期处理函数和操作符

6.5.1 时间/日期操作符

须知

用户在使用时间和日期操作符时，对应的操作数请使用明确的类型前缀修饰，以确保数据库在解析操作数的时候能够与用户预期一致，不会产生用户非预期的结果。

比如下面示例没有明确数据类型就会出现异常错误。

```
SELECT date '2001-10-01' - '7' AS RESULT;
ERROR:  invalid input syntax for type timestamp: "7"
```

表 6-2 时间和日期操作符

操作符	示例

+	date类型参数与integer参数相加，获取时间间隔为7天后的时间：
	`SELECT date '2001-09-28' + integer '7' AS RESULT;`
	`      result`
	`---------------------`
	` 2001-10-05 00:00:00`
	`(1 row)`

	date类型参数与interval参数相加，获取时间间隔为1小时后的时间：
	`SELECT date '2001-09-28' + interval '1 hour' AS RESULT;`
	`      result`
	`---------------------`
	` 2001-09-28 01:00:00`
	`(1 row)`

	date类型参数与time类型参数相加，获取具体的日期和时间结果：
	`SELECT date '2001-09-28' + time '03:00' AS RESULT;`
	`      result`
	`---------------------`
	` 2001-09-28 03:00:00`
	`(1 row)`

	date类型参数与interval参数相加，获取时间间隔为1个月的时间：
	date函数对于日期相加减超过月份的日期范围，会对齐到对应月份最后一天，不超过则不处理。例如：2021-01-31后一个月的日期为2021-02-31，但2月为闰月，只有28天，那么date函数会返回对齐到2月份最后一天的结果，即2021-02-28。
	`SELECT date '2021-01-31' + interval '1 month' AS RESULT;`
	`      result`
	`---------------------`
	` 2021-02-28 00:00:00`
	`(1 row)`
	`SELECT date '2021-02-28' + interval '1 month' AS RESULT;`
	`      result`
	`---------------------`
	` 2021-03-28 00:00:00`
	`(1 row)`

	interval参数相加，获取两个时间间隔之和：
	`SELECT interval '1 day' + interval '1 hour' AS RESULT;`
	`    result`
	`----------------`
	` 1 day 01:00:00`
	`(1 row)`

操作符	示例
	timestamp时间类型参数与interval参数相加，获取间隔23小时后的时间： SELECT timestamp '2001-09-28 01:00' + interval '23 hours' AS RESULT; result --- 2001-09-29 00:00:00 (1 row)
	time类型参数与interval参数相加，获取间隔时间为3小时后的时间： SELECT time '01:00' + interval '3 hours' AS RESULT; result --- 04:00:00 (1 row)
-	date类型参数相减，获取两个日期的时间差： SELECT date '2001-10-01' - date '2001-09-28' AS RESULT; result --- 3 days (1 row)
	date类型参数与integer参数相减，返回timestamp类型，获取两者的时间差： SELECT date '2001-10-01' - integer '7' AS RESULT; result --- 2001-09-24 00:00:00 (1 row)
	date类型参数与interval参数相减，获取两者的日期、时间差： SELECT date '2001-09-28' - interval '1 hour' AS RESULT; result --- 2001-09-27 23:00:00 (1 row)
	time类型参数相减，获取两参数的时间差： SELECT time '05:00' - time '03:00' AS RESULT; result --- 02:00:00 (1 row)
	time类型参数与interval相减，获取两参数的时间差： SELECT time '05:00' - interval '2 hours' AS RESULT; result --- 03:00:00 (1 row)
	timestamp类型参数与interval相减，从时间戳中减去时间间隔，获取两者的日期时间差： SELECT timestamp '2001-09-28 23:00' - interval '23 hours' AS RESULT; result --- 2001-09-28 00:00:00 (1 row)

操作符	示例
	interval参数相减，获取两者的时间差： SELECT interval '1 day' - interval '1 hour' AS RESULT; result ---------- 23:00:00 (1 row)
	timestamp类型参数相减，获取两者的日期时间差： SELECT timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' AS RESULT; result ---------------- 1 day 15:00:00 (1 row)
	获取当前日期的前一天： SELECT now() - interval '1 day'AS RESULT; result ------------------------------- 2022-08-08 01:46:15.555406+00 (1 row)

*	将时间间隔乘以数量：
	SELECT 900 * interval '1 second' AS RESULT;
	  result
	---------- 	
	 00:15:00
	(1 row)
	SELECT 21 * interval '1 day' AS RESULT;
	 result
	--------- 	
	 21 days
	(1 row)
	SELECT double precision '3.5' * interval '1 hour' AS RESULT;
	  result
	---------- 	
	 03:30:00
	(1 row)

/	用时间间隔除以数量，获取一段时间中的某一段： SELECT interval '1 hour' / double precision '1.5' AS RESULT; result ---------- 00:40:00 (1 row)

6.5.2 时间/日期函数

age(timestamp, timestamp)

描述：将两个参数相减，并以年、月、日作为返回值。若相减值为负，则函数返回亦为负。

返回值类型：interval

示例：

```
SELECT age(TIMESTAMP '2001-04-10', TIMESTAMP '1957-06-13');
          age
```



```
-------------------------
 43 years 9 mons 27 days
(1 row)
```

age(timestamp)

描述：当前时间和参数相减。

返回值类型：interval

示例：

```
SELECT age(TIMESTAMP '1957-06-13');
          age
-------------------------
 60 years 2 mons 18 days
(1 row)
```

adddate(date, interval | int)

描述：返回给定日期时间加上指定单位的时间间隔的结果。默认单位(即第二个参数为整型时)为天数。

返回值类型：timestamp

示例：

当入参为text类型时：

```
SELECT adddate('2020-11-13', 10);
  adddate
------------
 2020-11-23
(1 row)

SELECT adddate('2020-11-13', interval '1' month);
  adddate
------------
 2020-12-13
(1 row)

SELECT adddate('2020-11-13 12:15:16', interval '1' month);
       adddate
---------------------
 2020-12-13 12:15:16
(1 row)


SELECT adddate('2020-11-13', interval '1' minute);
       adddate
---------------------
 2020-11-13 00:01:00
(1 row)
```

当入参为date类型时：

```
SELECT adddate(current_date, 10);
  adddate
------------
 2021-09-24
(1 row)

SELECT adddate(date '2020-11-13', interval '1' month);
       adddate
---------------------
 2020-12-13 00:00:00
(1 row)
```


subdate(date, interval | int)

描述：返回给定日期时间减去指定单位的时间间隔的结果; 默认单位(即第二个参数为整型时)为天数。

返回值类型：timestamp

示例：

当入参为text类型时：

```
SELECT subdate('2020-11-13', 10);
  subdate
------------
 2020-11-03
(1 row)

SELECT subdate('2020-11-13', interval '2' month);
  subdate
------------
 2020-09-13
(1 row)

SELECT subdate('2020-11-13 12:15:16', interval '1' month);
       subdate
---------------------
 2020-10-13 12:15:16
(1 row)

SELECT subdate('2020-11-13', interval '2' minute);
       subdate
---------------------
 2020-11-12 23:58:00
(1 row)
```

当入参为date类型时：

```
SELECT subdate(current_date, 10);
  subdate
------------
 2021-09-05
(1 row)

SELECT subdate(current_date, interval '1' month);
       subdate
---------------------
 2021-08-15 00:00:00
(1 row)
```

date_add(date, interval)

描述：返回给定日期时间加上指定单位的时间间隔的结果。等效于**adddate(date, interval | int)**。

返回值类型：timestamp

date_sub(date, interval)

描述：返回给定日期时间减去指定单位的时间间隔的结果，等效于**subdate(date, interval | int)**。

返回值类型：timestamp

timestampadd(field, numeric, timestamp)

描述：将以单位field的整数时间间隔（秒数可以带小数）添加到日期时间表达式中。若数值为负，则表示从给定的时间日期时间表达式中减去对应的时间间隔。field支持的参数为year，month，quarter，day，week，hour，minute，second和microsecond。

返回值类型：timestamp

示例：

```
SELECT timestampadd(year, 1, TIMESTAMP '2020-2-29');
    timestampadd
---------------------
 2021-02-28 00:00:00
(1 row)

SELECT timestampadd(second, 2.354156, TIMESTAMP '2020-11-13');
         timestampadd
----------------------------
 2020-11-13 00:00:02.354156
(1 row)
```

timestampdiff(field, timestamp1, timestamp2)

描述：将两个日期参数相减(timestamp2 - timestamp1)，并以单位field作为返回值。若相减值为负，则函数返回值为负。field支持的参数为year、month、quarter、day、week、hour、minute、second和microsecond。

返回值类型：bigint

示例：

```
SELECT timestampdiff(day, TIMESTAMP '2001-02-01', TIMESTAMP '2003-05-01 12:05:55');
 timestampdiff
---------------
      819
(1 row)
```

clock_timestamp()

描述：实时时钟的当前时间戳。

返回值类型：timestamp with time zone

示例：

```
SELECT clock_timestamp();
         clock_timestamp
-------------------------------
 2017-09-01 16:57:36.636205+08
(1 row)
```

current_date

描述：当前日期。

返回值类型：date

示例：

```
SELECT current_date;
    date
```



```
------------
 2017-09-01
(1 row)
```

current_time

描述：当前时间。

返回值类型：time with time zone

示例：

```
SELECT current_time;
      timetz
--------------------
 16:58:07.086215+08
(1 row)
```

current_timestamp

描述：当前日期及时间。

返回值类型：timestamp with time zone

示例：

```
SELECT current_timestamp;
      pg_systimestamp
-------------------------------
 2017-09-01 16:58:19.22173+08
(1 row)
```

datediff(date1, date2)

描述：返回给定日期之间相差的天数值。

返回值类型：integer

示例：

```
SELECT datediff(date '2020-11-13', date '2012-10-16');
 datediff
----------
     2950
(1 row)
```

date_part(text, timestamp)

描述：获取参数text指定的精度。

等效于**extract(field from timestamp)**。

返回值类型：double precision

示例：

```
SELECT date_part('hour', TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

date_part(text, interval)

描述：获取参数text指定的精度。如果大于12，则取与12的模。

等效于**extract(field from timestamp)**。

返回值类型：double precision

示例：

```
SELECT date_part('month', interval '2 years 3 months');
 date_part
-----------
         3
(1 row)
```

date_trunc(text, timestamp)

描述：截取到参数text指定的精度。

返回值类型：timestamp

示例：

```
SELECT date_trunc('hour', TIMESTAMP  '2001-02-16 20:38:40');
     date_trunc
---------------------
 2001-02-16 20:00:00
(1 row)

--获取去年的最后一天
SELECT date_trunc('day', date_trunc('year',CURRENT_DATE)+ '-1');
       date_trunc
-------------------------
 2022-12-31 00:00:00+00
(1 row)

--获取今年的第一天
SELECT date_trunc('year',CURRENT_DATE);
       date_trunc
-------------------------
 2023-01-01 00:00:00+00
(1 row)

--获取去年的第一天
SELECT date_trunc('year',now() + '-1 year');
       date_trunc
-------------------------
 2022-01-01 00:00:00+00
(1 row)
```

trunc(timestamp)

描述：默认按天截取。

返回值类型：timestamp

示例：

```
SELECT trunc(TIMESTAMP  '2001-02-16
20:38:40');
                trunc
---------------------
2001-02-16 00:00:00
(1 row)
```

extract(field from timestamp)

描述：获取field指定精度的值。field的有效值参见**EXTRACT**。

返回值类型：double precision

示例：

```
SELECT extract(hour FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

extract(field from interval)

描述：获取field指定精度的值。如果大于12，则取与12的模。field的有效值参见**EXTRACT**。

返回值类型：double precision

示例：

```
SELECT extract(month FROM interval '2 years 3 months');
 date_part
-----------
         3
(1 row)
```

day(date)

描述：获取日期时间date所处月份中的天数，与dayofmonth函数相同。

取值范围：1~31

返回值类型：integer

示例：

```
SELECT day('2020-06-28');
 day
-----
  28
(1 row)
```

dayofmonth(date)

描述：获取日期时间date所处月份中的天数。

取值范围：1~31

返回值类型：integer

示例：

```
SELECT dayofmonth('2020-06-28');
 dayofmonth
------------
         28
(1 row)
```

dayofweek(date)

描述：返回给定日期date对应的星期索引，星期日作为一周的开始日。

取值范围：1~7

返回值类型：integer

示例：

```
SELECT dayofweek('2020-11-22');
 dayofweek
-----------
         1
(1 row)
```

dayofyear(date)

描述：返回给定日期date在本年中的天数。

取值范围：1~366

返回值类型：integer

示例：

```
SELECT dayofyear('2020-02-29');
 dayofyear
-----------
        60
(1 row)
```

hour(timestamp with time zone)

描述：获取时间中的小时值。

返回值类型：integer

示例：

```
SELECT hour(timestamptz '2018-12-13 12:11:15+06');
 hour
------
    6
(1 row)
```

isfinite(date)

描述：测试是否为有限日期。

返回值类型：boolean

示例：

```
SELECT isfinite(date '2001-02-16');
 isfinite
----------
 t
(1 row)
SELECT isfinite(date 'infinity');
 isfinite
----------
 f
(1 row)
```

isfinite(timestamp)

描述：测试判断是否为有限时间。

返回值类型：boolean

示例：

```
SELECT isfinite(timestamp '2001-02-16 21:28:30');
 isfinite
----------
 t
(1 row)
SELECT isfinite(timestamp 'infinity');
 isfinite
----------
 f
(1 row)
```

isfinite(interval)

描述：测试是否为有限时间间隔。

返回值类型：boolean

示例：

```
SELECT isfinite(interval '4 hours');
 isfinite
----------
 t
(1 row)
```

justify_days(interval)

描述：将时间间隔以30天为单位，表示为月。

返回值类型：interval

示例：

```
SELECT justify_days(interval '35 days');
 justify_days
--------------
 1 mon 5 days
(1 row)
```

justify_hours(interval)

描述：将时间间隔以24小时为单位，表示为天。

返回值类型：interval

示例：

```
SELECT justify_hours(interval '27 HOURS');
 justify_hours
----------------
 1 day 03:00:00
(1 row)
```

justify_interval(interval)

描述：结合justify_days和justify_hours，调整interval。

返回值类型：interval

示例：


```
SELECT justify_interval(interval '1 MON -1 HOUR');
 justify_interval
------------------
 29 days 23:00:00
(1 row)
```

localtime

描述：当前时间。

返回值类型：time

示例：

```
SELECT localtime ;
     time
----------------
 16:05:55.664681
(1 row)
```

localtimestamp

描述：当前日期及时间。

返回值类型：timestamp

示例：

```
SELECT localtimestamp;
         timestamp
----------------------------
 2017-09-01 17:03:30.781902
(1 row)
```

makedate(year, dayofyear)

描述：根据给定的年份和一年中的天数返回相对应的日期值。

返回值类型：date

示例：

```
SELECT makedate(2020, 60);
  makedate
------------
 2020-02-29
(1 row)
```

maketime(hour, minute, second)

描述：根据所给的小时，分钟和秒数返回time类型的值。由于GaussDB(DWS)中的time类型的取值范围为00:00:00到24:00:00, 故不支持hour大于24时和hour小于0时的场景。

返回值类型：time

示例：

```
SELECT maketime(12, 15, 30.12);
  maketime
-------------
 12:15:30.12
(1 row)
```


microsecond(timestamp with time zone)

描述：获取时间中的微秒值。

返回值类型：integer

示例：

```
SELECT microsecond(timestamptz '2018-12-13 12:11:15.123634+06');
 microsecond
-------------
      123634
(1 row)
```

minute(timestamp with time zone)

描述：获取时间中的分钟值。

返回值类型：integer

示例：

```
SELECT minute(timestamptz '2018-12-13 12:11:15+06');
 minute
--------
     11
(1 row)
```

month(date)

描述：返回给定日期时间的月份。

返回值类型：integer

示例：

```
SELECT month('2020-11-30');
 month
-------
    11
(1 row)
```

now([fsp])

描述：当前事务开始的日期及时间，参数确定微秒输出精度，缺省时为6。

返回值类型：timestamp with time zone

示例：

```
SELECT now();
               now
-------------------------------
 2017-09-01 17:03:42.549426+08
(1 row)
SELECT now(3);
              now
----------------------------
 2021-09-08 10:59:00.427+08
(1 row)
```


numtodsinterval(num, interval_unit)

描述：将数字转换为interval类型。num为numeric类型数字，interval_unit为固定格式字符串（ 'DAY' | 'HOUR' | 'MINUTE' | 'SECOND' ）。

可以通过设置参数IntervalStyle为oracle，兼容该函数在Oracle中的interval输出格式。

示例：

```
SELECT numtodsinterval(100, 'HOUR');
 numtodsinterval
-----------------
 100:00:00
(1 row)

SET intervalstyle = oracle;
SET
SELECT numtodsinterval(100, 'HOUR');
        numtodsinterval
-------------------------------
 +000000004 04:00:00.000000000
(1 row)
```

pg_sleep(seconds)

描述：将当前会话的进行暂停指定的秒数。

返回值类型：void

示例：

```
SELECT pg_sleep(10);
 pg_sleep
----------

(1 row)
```

period_add(P, N)

描述：返回给定时期加上N个月后的日期。

返回值类型：integer

示例：

```
SELECT period_add(200801, 2);
 period_add
------------
     200803
(1 row)
```

period_diff(P1, P2)

描述：返回给定日期之间的月数差值。

返回值类型：integer

```
SELECT period_diff(200802, 200703);
 period_diff
-------------
          11
(1 row)
```


quarter(date)

描述：获取日期date所属的季度。

返回值类型：integer

示例：

```
SELECT quarter(date '2018-12-13');
 quarter
---------
       4
(1 row)
```

second(timestamp with time zone)

描述：获取时间的秒数值。

返回值类型：integer

示例：

```
SELECT second(timestamptz '2018-12-13 12:11:15+06');
 second
--------
     15
(1 row)
```

statement_timestamp()

描述：当前日期及时间。

返回值类型：timestamp with time zone

示例：

```
SELECT statement_timestamp();
      statement_timestamp
-------------------------------
 2017-09-01 17:04:39.119267+08
(1 row)
```

sysdate

描述：当前日期及时间。

返回值类型：timestamp

示例：

```
SELECT sysdate;
       sysdate
---------------------
 2017-09-01 17:04:49
(1 row)
```

timeofday()

描述：当前日期及时间（像clock_timestamp，但是返回时为text。）

返回值类型：text

示例：


```
SELECT timeofday();
              timeofday
-------------------------------------
 Fri Sep 01 17:05:01.167506 2017 CST
(1 row)
```

transaction_timestamp()

描述：当前日期及时间，与current_timestamp等效。

返回值类型：timestamp with time zone

示例：

```
SELECT transaction_timestamp();
    transaction_timestamp
-------------------------------
 2017-09-01 17:05:13.534454+08
(1 row)
```

from_unixtime(unix_timestamp[,format])

描述：格式串缺省时，将unix时间戳转换为日期时间类型输出。格式串指定时，将unix时间戳转换为指定格式的字符串输出。

返回值类型：timestamp（格式串缺省）/ text（格式串指定）

示例：

```
SELECT from_unixtime(875996580);
   from_unixtime
---------------------
 1997-10-04 20:23:00
(1 row)
SELECT from_unixtime(875996580, '%Y %D %M %h:%i:%s');
      from_unixtime
---------------------------
 1997 4th October 08:23:00
(1 row)
```

unix_timestamp([timestamp with time zone])

描述：获取从'1970-01-01 00:00:00'UTC到入参时间经历的秒数。无入参时，指定为当前时间。

返回值类型：bigint（无入参）/numeric（有入参）

示例：

```
SELECT unix_timestamp();
 unix_timestamp
----------------
    1693906219
(1 row)
SELECT unix_timestamp('2018-09-08 12:11:13+06');
 unix_timestamp
----------------
    1536387073.000000
(1 row)
```

add_months(d,n)

描述：用于计算时间点d再加上n个月的时间。

返回值类型：timestamp

示例：

```
SELECT add_months(to_date('2017-5-29', 'yyyy-mm-dd'), 11);
    add_months
---------------------
 2018-04-29 00:00:00
(1 row)
```

last_day(d)

描述：用于计算时间点d本月最后一天的时间。

- ORA和TD兼容模式下，返回值类型为timestamp。
- MySQL兼容模式下，返回值类型为date。

示例：

```
SELECT last_day(to_date('2017-01-01', 'YYYY-MM-DD')) AS cal_result;
    cal_result
---------------------
 2017-01-31 00:00:00
(1 row)
```

next_day(x,y)

描述：用于计算x时间开始的下一个星期y的时间。

- ORA和TD兼容模式下，返回值类型为timestamp。
- MySQL兼容模式下，返回值类型为date。

示例：

```
SELECT next_day(TIMESTAMP '2017-05-25 00:00:00','Sunday')AS cal_result;
    cal_result
---------------------
 2017-05-28 00:00:00
(1 row)
```

from_days(days)

描述：根据给定的天数，返回相对应的日期值。

返回值类型：date

示例：

```
SELECT from_days(730669);
 from_days
------------
 2000-07-03
(1 row)
```

to_days(timestamp)

描述：返回自0年开始到入参日期的天数。

返回值类型：integer

示例：


```
SELECT to_days(TIMESTAMP '2008-10-07');
 to_days
---------
  733687
(1 row)
```

6.5.3 EXTRACT

EXTRACT(*field* **FROM** *source***)**

extract函数从日期或时间的数值里抽取子域，比如年、小时等。source必须是一个timestamp、time或interval类型的值表达式（类型为date的表达式转换为timestamp，因此也可以用）。field是一个标识符或者字符串，它指定从源数据中抽取的域。extract函数返回类型为double precision的数值。field的取值范围如下所示。

century

世纪。

第一个世纪从0001-01-01 00:00:00 AD开始。这个定义适用于所有使用阳历的国家。没有0世纪，直接从公元前1世纪到公元1世纪。

示例：

```
SELECT EXTRACT(century FROM TIMESTAMP '2000-12-16 12:21:13');
 date_part
-----------
        20
(1 row)
```

day

- 如果source为timestamp，表示月份里的日期（1-31）。

  ```
  SELECT EXTRACT(day FROM TIMESTAMP '2001-02-16 20:38:40');
   date_part
  -----------
         16
  (1 row)
  ```

- 如果source为interval，表示天数。

  ```
  SELECT EXTRACT(day FROM INTERVAL '40 days 1 minute');
   date_part
  -----------
         40
  (1 row)
  ```

decade

年份除以10。

```
SELECT EXTRACT(decade FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
       200
(1 row)
```

dow

每周的星期几，星期天（0）到星期六（6）。

```
SELECT EXTRACT(dow FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
```



```
       5
(1 row)
```

doy

一年的第几天（1~365/366）。

```
SELECT EXTRACT(doy FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        47
(1 row)
```

epoch

- 如果source为timestamp with time zone，表示自1970-01-01 00:00:00-00 UTC以来的秒数（结果可能是负数）；

  如果source为date和timestamp，表示自1970-01-01 00:00:00-00当地时间以来的秒数；

  如果source为interval，表示时间间隔的总秒数。

  ```
  SELECT EXTRACT(epoch FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
    date_part
  ---------------
   982384720.12
  (1 row)
  SELECT EXTRACT(epoch FROM interval '5 days 3 hours');
   date_part
  -----------
      442800
  (1 row)
  ```

- 将epoch值转换为时间戳的方法。

  ```
  SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * interval '1 second' AS RESULT;
            result
  ----------------------------
   2001-02-17 12:38:40.12+08
  (1 row)
  ```

hour

小时域（0-23）。

```
SELECT EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

isodow

一周的第几天（1-7）。

星期一为1，星期天为7。

说明

除了星期天外，都与dow相同。

```
SELECT EXTRACT(isodow FROM TIMESTAMP '2001-02-18 20:38:40');
 date_part
-----------
         7
(1 row)
```


isoyear

日期中的ISO 8601标准年（不适用于间隔）。

每个带有星期一开始的周中包含1月4日的ISO年，所以在年初的1月或12月下旬的ISO年可能会不同于阳历的年。详细信息请参见后续的**week**描述。

```
SELECT EXTRACT(isoyear FROM DATE '2006-01-01');
 date_part
-----------
      2005
(1 row)
SELECT EXTRACT(isoyear FROM DATE '2006-01-02');
 date_part
-----------
      2006
(1 row)
```

microseconds

秒域（包括小数部分）乘以1,000,000。

```
SELECT EXTRACT(microseconds FROM TIME '17:12:28.5');
 date_part
-----------
  28500000
(1 row)
```

millennium

千年。

20世纪（19xx年）里面的年份在第二个千年里。第三个千年从2001年1月1日零时开始。

```
SELECT EXTRACT(millennium FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         3
(1 row)
```

milliseconds

秒域（包括小数部分）乘以1000。请注意它包括完整的秒。

```
SELECT EXTRACT(milliseconds FROM TIME '17:12:28.5');
 date_part
-----------
     28500
(1 row)
```

minute

分钟域（0-59）。

```
SELECT EXTRACT(minute FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        38
(1 row)
```

month

如果source为timestamp，表示一年里的月份数（1-12）。

```
SELECT EXTRACT(month FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         2
(1 row)
```

如果source为interval，表示月的数目，然后对12取模（0-11）。

```
SELECT EXTRACT(month FROM interval '2 years 13 months');
 date_part
-----------
         1
(1 row)
```

quarter

该天所在的该年的季度（1-4）。

```
SELECT EXTRACT(quarter FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         1
(1 row)
```

second

秒域，包括小数部分（0-59）。

```
SELECT EXTRACT(second FROM TIME '17:12:28.5');
 date_part
-----------
      28.5
(1 row)
```

timezone

与UTC的时区偏移量，单位为秒。正数对应UTC东边的时区，负数对应UTC西边的时区。

```
SELECT EXTRACT(timezone FROM TIMETZ '17:12:28');
 date_part
-----------
    0
(1 row)
```

timezone_hour

时区偏移量的小时部分。

```
SELECT EXTRACT(timezone_hour FROM TIMETZ '17:12:28');
 date_part
-----------
       0
(1 row)
```

timezone_minute

时区偏移量的分钟部分。


```
SELECT EXTRACT(timezone_minute FROM TIMETZ '17:12:28');
 date_part
-----------
         0
(1 row)
```

week

该天在所在的年份里是第几周。ISO 8601定义一年的第一周包含该年的一月四日（ISO-8601 的周从星期一开始）。换句话说，一年的第一个星期四在第一周。

在ISO定义里，一月的头几天可能是前一年的第52或者第53周，十二月的后几天可能是下一年第一周。比如，2005-01-01是2004年的第53周，而2006-01-01是2005年的第52周，2012-12-31是2013年的第一周。建议isoyear字段和week一起使用以得到一致的结果。

```
SELECT EXTRACT(week FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
         7
(1 row)
```

year

年份域。

```
SELECT EXTRACT(year FROM TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
      2001
(1 row)
```

6.5.4 date_part

date_part函数是在传统的Ingres函数的基础上制作的（该函数等效于SQL标准函数extract）：

```
date_part('field', source)
```

这里的field参数必须是一个字符串，而不是一个名字。有效的field与extract一样，详细信息请参见**EXTRACT**。

示例：

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
 date_part
-----------
        16
(1 row)
SELECT date_part('hour', interval '4 hours 3 minutes');
 date_part
-----------
         4
(1 row)
```

6.5.5 date_format

date_format(timestamp, fmt)

date_format函数将日期参数按照fmt指定的格式转换为字符串。

示例：


```
SELECT date_format('2009-10-04 22:23:00', '%M %D %W');
   date_format
-------------------
 October 4th Sunday
(1 row)
SELECT date_format('2021-02-20 08:30:45', '%Y-%m-%d %H:%i:%S');
    date_format
--------------------
 2021-02-20 08:30:45
(1 row)
SELECT date_format('2021-02-20 18:10:15', '%r-%T');
    date_format
----------------------
 06:10:15 PM-18:10:15
(1 row)
```

表 date_format支持的输出格式显示了可以用于将日期参数格式化输出的格式类型，这些格式类型适用于函数date_format、time_format、str_to_date、str_to_time和from_unixtime。

表 6-3 date_format 支持的输出格式

格式	说明	取值
%a	缩写星期名	Sun...Sat
%b	缩写月份名	Jan...Dec
%c	月份	0...12
%D	带英文后缀的月份日期	0th, 1st, 2nd, 3rd, …
%d	一个月里的日，2位	00...31
%e	一个月里的日	0...31
%f	微秒	000000...999999
%H	小时，24小时制	00...23
%h	小时，12小时制	01...12
%I	小时，12小时制，同%h	01...12
%i	分钟	00...59
%j	一年里的日	001...366
%k	小时，24小时制，同%H	0...23
%l	小时，12小时制，同%h	1...12
%M	月份名	January...December
%m	月份，两位	00...12
%p	上下午	AM PM
%r	时间，12小时制	hh::mm::ss AM/PM
%S	秒	00...59
%s	秒，同%S	00...59

格式	说明	取值
%T	时间，24小时制	hh::mm::ss
%U	周 (00-53) 星期日是一周的第一天	00...53
%u	周 (00-53) 星期一是一周的第一天	00...53
%V	周 (01-53) 星期日是一周的第一天，与%X搭配使用	01...53
%v	周 (01-53) 星期一是一周的第一天，与%x搭配使用	01...53
%W	星期名	Sunday...Saturday
%w	一周的日，周日为0	0...6
%X	年份，其中的星期日是周的第一天，4 位，与%V搭配使用	-
%x	年份，其中的星期一是周的第一天，4 位，与%v搭配使用	-
%Y	年份，4位	-
%y	年份，2位	-
%%	字符'%'	字符'%'
%*x*	'*x*'，上述未列出的任意字符	字符'*x*'

须知

date_format支持的输出格式中，%U、%u、%V、%v、%X、%x暂不支持。

6.5.6 time_format

time_format(time, fmt)

描述：time_format函数将日期参数按照fmt指定的格式转换为字符串。与date_format函数类似，但格式字符串只能包含小时、分钟、秒和微秒的格式说明符，如果包含其他说明符则会返回NULL值或0。

返回值类型：text

示例：

```
SELECT time_format('2009-10-04 22:23:00', '%M %D %W');
   time_format
--------------------

(1 row)
SELECT time_format('2021-02-20 08:30:45', '%Y-%m-%d %H:%i:%S');
```

```
    time_format
---------------------
 0000-00-00 08:30:45
(1 row)
SELECT time_format('2021-02-20 18:10:15', '%r-%T');
    time_format
----------------------
 06:10:15 PM-18:10:15
(1 row)
```

须知

time_format仅支持时间相关的格式输出（%f、%H、%h、%I、%i、%k、%l、%p、%r、%S、%s、%T），不支持日期相关格式，其他情况处理为普通字符。

str_to_date(str, format)

描述：将日期/时间格式的字符串（str），按照所提供的显示格式（format）转换为日期类型的值。

返回值类型：timestamp

示例：

```
SELECT str_to_date('01,5,2021','%d,%m,%Y');
    str_to_date
---------------------
 2021-05-01 00:00:00
(1 row)
SELECT str_to_date('01,5,2021,09,30,17','%d,%m,%Y,%h,%i,%s');
    str_to_date
---------------------
 2021-05-01 09:30:17
(1 row)
```

适用于str_to_date的格式化输入的格式类型参考**表6-3**。这里仅支持“日期”格式、“日期+时间”格式的输入转换，对于仅“时间”格式的输入场景请使用str_to_time。

str_to_time(str, format)

描述：将时间格式的字符串（str），按照所提供的显示格式（format）转换为时间类型的值。

返回值类型：time

示例：

```
SELECT str_to_time('09:30:17','%h:%i:%s');
 str_to_time
-------------
 09:30:17
(1 row)
```

适用于str_to_time的格式化输入的格式类型参考**表6-3**，这里仅支持“时间”格式的输入转换，对于“日期”格式、“日期+时间”格式的输入场景请使用str_to_date。

week(date[, mode])

描述：根据模式返回指定日期时间所处年份中对应的周数，默认模式为0。

返回值类型：integer

表 6-4 week 函数中 mode 模式的工作原理

模式	一周的第一天	周数范围	第一周的判断规则
0	星期日	0-53	元旦后的第一个星期日所在周
1	星期一	0-53	元旦后有四天或者更多天所在周
2	星期日	1-53	元旦后的第一个星期日所在周
3	星期一	1-53	元旦后有四天或者更多天所在周
4	星期日	0-53	元旦后有四天或者更多天所在周
5	星期一	0-53	元旦后的第一个星期一所在周
6	星期日	1-53	元旦后有四天或者更多天所在周
7	星期一	1-53	元旦后的第一个星期一所在周

示例：

```
SELECT week('2018-01-01');
 week
------
    0
(1 row)

SELECT week('2018-01-01', 0);
 week
------
    0
(1 row)

SELECT week('2020-12-31', 1);
 week
------
   53
(1 row)

SELECT week('2020-12-31', 5);
 week
------
   52
(1 row)
```

weekday(date)

描述：返回给定日期date对应的星期索引，星期一作为一周的开始日。

取值范围：0~6

返回值类型：integer

示例：

```
SELECT weekday('2020-11-06');
 weekday
---------
       4
(1 row)
```

weekofyear(date)

描述：返回给定日期date所在周在本年中对应的周数，取值范围为[1, 53]，等价于week(date, 3)。

返回值类型：integer

示例：

```
SELECT weekofyear('2020-12-30');
 weekofyear
------------
         53
(1 row)
```

year(date)

描述：获取时间日期date所处的年份

返回值类型：integer

示例：

```
SELECT year('2020-11-13');
 year
------
 2020
(1 row)
```

yearweek(date[, mode])

描述：返回给定日期date在本年中对应的年份和周数，周数范围为[1, 53]。

返回值类型：integer

示例：

```
SELECT yearweek('2019-12-31');
 yearweek
----------
   201952
(1 row)

SELECT yearweek('2019-1-1');
 yearweek
----------
   201852
(1 row)
```

6.6 SEQUENCE 函数

序列函数为用户从序列对象中获取后续的序列值提供了简单的多用户安全的方法。

说明

实时数仓（单机部署）暂不支持SEQUENCE及相关函数。

nextval(regclass)

递增序列并返回新值。

说明

- 为了避免从同一个序列获取值的并发事务被阻塞， nextval操作不会回滚；也就是说，一旦一个值已经被抓取， 那么就认为它已经被用过了，并且不会再被返回。 即使该操作处于事务中，当事务之后中断，或者如果调用查询结束不使用该值，也是如此。这种情况将在指定值的顺序中留下未使用的"空洞"。 因此，GaussDB(DWS)序列对象不能用于获得"无间隙"序列。
- 如果nextval被下推到DN上时，各个DN会自动连接GTM，请求next values值，例如（insert into t1 select xxx，t1某一列需要调用nextval函数），由于GTM上有最大连接数为8192的限制，而这类下推语句会导致消耗过多的GTM连接数，因此对于这类语句的并发数目限制为7000（其它语句需要占用部分连接）/集群DN数目。

返回类型：bigint

nextval函数有两种调用方式（其中第二种调用方式兼容Oracle的语法，目前不支持Sequence命名中有特殊字符"."的情况），如下：

示例1：

```
SELECT nextval('seqDemo');
 nextval
---------
       2
(1 row)
```

示例2：

```
SELECT seqDemo.nextval;
 nextval
---------
       2
(1 row)
```

currval(regclass)

返回当前会话里最近一次nextval返回的指定的sequence的数值。如果当前会话还没有调用过指定的sequence的nextval，那么调用currval将会报错。需要注意的是，这个函数在默认情况下是不支持的，需要通过设置enable_beta_features为true之后，才能使用这个函数。同时在设置enable_beta_features为true之后，nextval()函数将不支持下推。

返回类型：bigint

currval函数有两种调用方式（其中第二种调用方式兼容Oracle的语法，目前不支持Sequence命名中有特殊字符"."的情况），如下：

示例1：

```
SELECT currval('seq1');
 currval
---------
       2
(1 row)
```

示例2：

```
SELECT seq1.currval seq1;
 currval
---------
       2
(1 row)
```


lastval()

描述：返回当前会话里最近一次nextval返回的数值。这个函数等效于currval，只是它不用序列名为参数，它抓取当前会话里面最近一次nextval使用的序列。如果当前会话还没有调用过nextval，那么调用lastval将会报错。

需要注意的是，这个函数在默认情况下是不支持的，需要通过设置enable_beta_features或者lastval_supported为true之后，才能使用这个函数。同时这种情况下，nextval()函数将不支持下推。

返回类型：bigint

示例：

```
SELECT lastval();
 lastval
---------
       2
(1 row)
```

setval(regclass, bigint)

描述：设置序列的当前数值。

返回类型：bigint

示例：

```
SELECT setval('seqDemo',1);
 setval
--------
      1
(1 row)
```

setval(regclass, bigint, boolean)

描述：设置序列的当前数值以及is_called标志。

返回类型：bigint

示例：

```
SELECT setval('seqDemo',1,true);
 setval
--------
      1
(1 row)
```

说明

Setval后当前会话及GTM上会立刻生效，但如果其他会话有缓存的序列值，只能等到缓存值用尽才能感知Setval的作用。所以为了避免序列值冲突，setval要谨慎使用。

因为序列是非事务的，setval造成的改变不会由于事务的回滚而撤销。

6.7 数组函数和操作符

6.7.1 数组操作符

数组比较是使用默认的B-tree比较函数对所有元素逐一进行比较的。多维数组的元素按照行顺序进行访问。如果两个数组的内容相同但维数不等，决定排序顺序的首要因素是维数。

=

描述：两个数组是否相等

示例：

```
SELECT ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] AS RESULT;
 result
--------
 t
(1 row)
```

<>

描述：两个数组是否不相等

示例：

```
SELECT ARRAY[1,2,3] <> ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

<

描述：一个数组是否小于另一个数组

示例：

```
SELECT ARRAY[1,2,3] < ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

>

描述：一个数组是否大于另一个数组

示例：

```
SELECT ARRAY[1,4,3] > ARRAY[1,2,4] AS RESULT;
 result
--------
 t
(1 row)
```

<=

描述：一个数组是否小于或等于另一个数组

示例：

```
SELECT ARRAY[1,2,3] <= ARRAY[1,2,3] AS RESULT;
 result
```



```
--------
 t
(1 row)
```

>=

描述：一个数组是否大于或等于另一个数组

示例：

```
SELECT ARRAY[1,4,3] >= ARRAY[1,4,3] AS RESULT;
 result
--------
 t
(1 row)
```

@>

描述：一个数组是否包含另一个数组

示例：

```
SELECT ARRAY[1,4,3] @> ARRAY[3,1] AS RESULT;
 result
--------
 t
(1 row)
```

<@

描述：一个数组是否被包含于另一个数组

示例：

```
SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6] AS RESULT;
 result
--------
 t
(1 row)
```

&&

描述：一个数组是否和另一个数组重叠（有共同元素）

示例：

```
SELECT ARRAY[1,4,3] && ARRAY[2,1] AS RESULT;
 result
--------
 t
(1 row)
```

||

描述：数组与元素交叉组合进行连接，即数组与数组进行连接、元素与数组进行连接、数组与元素进行连接。

示例：

```
SELECT ARRAY[1,2,3] || ARRAY[4,5,6] AS RESULT;
    result
---------------
 {1,2,3,4,5,6}
(1 row)
```



```
SELECT ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] AS RESULT;
          result
---------------------------
 {{1,2,3},{4,5,6},{7,8,9}}
(1 row)
SELECT 3 || ARRAY[4,5,6] AS RESULT;
  result
-----------
 {3,4,5,6}
(1 row)
SELECT ARRAY[4,5,6] || 7 AS RESULT;
  result
-----------
 {4,5,6,7}
(1 row)
```

6.7.2 数组函数

array_append(anyarray, anyelement)

描述：向数组末尾添加元素，只支持一维数组。

返回类型：anyarray

示例：

```
SELECT array_append(ARRAY[1,2], 3) AS RESULT;
 result
---------
 {1,2,3}
(1 row)
```

array_prepend(anyelement, anyarray)

描述：向数组开头添加元素，只支持一维数组。

返回类型：anyarray

示例：

```
SELECT array_prepend(1, ARRAY[2,3]) AS RESULT;
 result
---------
 {1,2,3}
(1 row)
```

array_cat(anyarray, anyarray)

描述：连接两个数组，支持多维数组。

返回类型：anyarray

示例：

```
SELECT array_cat(ARRAY[1,2,3], ARRAY[4,5]) AS RESULT;
   result
-------------
 {1,2,3,4,5}
(1 row)

SELECT array_cat(ARRAY[[1,2],[4,5]], ARRAY[6,7]) AS RESULT;
       result
---------------------
 {{1,2},{4,5},{6,7}}
(1 row)
```


array_ndims(anyarray)

描述：返回数组的维数。

返回类型：int

示例：

```
SELECT array_ndims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
 result
--------
      2
(1 row)
```

array_dims(anyarray)

描述：返回数组维数的文本表示。

返回类型：text

示例：

```
SELECT array_dims(ARRAY[[1,2,3], [4,5,6]]) AS RESULT;
   result
------------
 [1:2][1:3]
(1 row)
```

array_length(anyarray, int)

描述：返回数组维度的长度。

返回类型：int

示例：

```
SELECT array_length(array[1,2,3], 1) AS RESULT;
 result
--------
      3
(1 row)
```

array_lower(anyarray, int)

描述：返回数组维数的下界。

返回类型：int

示例：

```
SELECT array_lower('[0:2]={1,2,3}'::int[], 1) AS RESULT;
 result
--------
      0
(1 row)
```

array_upper(anyarray, int)

描述：返回数组维数的上界。

返回类型：int

示例：


```
SELECT array_upper(ARRAY[1,8,3,7], 1) AS RESULT;
 result
--------
      4
(1 row)
```

array_to_string(anyarray, text [, text])

描述：使用第一个text作为数组的新分隔符，使用第二个text替换数组值为null的值。

返回类型：text

示例：

```
SELECT array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*') AS RESULT;
  result
-----------
 1,2,3,*,5
(1 row)
```

说明

在array_to_string中，如果省略null字符串参数或为NULL，运算中将跳过在数组中的任何null元素，并且不会在输出字符串中出现。

string_to_array(text, text [, text])

描述：使用第二个text指定分隔符，使用第三个可选的text作为NULL值替换模板，如果分隔后的子串与第三个可选的text完全匹配，则将其替换为NULL。

返回类型：text[]

示例：

```
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'yy') AS RESULT;
    result
--------------
 {xx,NULL,zz}
(1 row)
SELECT string_to_array('xx~^~yy~^~zz', '~^~', 'y') AS RESULT;
   result
------------
 {xx,yy,zz}
(1 row)
```

说明

- 在string_to_array中，如果分隔符参数是NULL，输入字符串中的每个字符将在结果数组中变成一个独立的元素。如果分隔符是一个空白字符串，则整个输入的字符串将变为一个元素的数组。否则输入字符串将在每个分隔字符串处分开。
- 在string_to_array中，如果省略null字符串参数或为NULL，将字符串中没有输入内容的子串替换为NULL。

unnest(anyarray)

描述：扩大一个数组为一组行。

返回类型：setof anyelement

示例：

```
SELECT unnest(ARRAY[1,2]) AS RESULT;
 result
--------
```



```
     1
     2
(2 rows)
```

unnest函数配合string_to_array数组使用。数组转列，先将字符串按逗号分割成数组，然后再把数组转成列：

```
SELECT unnest(string_to_array('a,b,c,d',',')) AS RESULT;
 result
--------
 a
 b
 c
 d
(4 rows)
```

6.8 逻辑操作符

常用的逻辑操作符有AND、OR和NOT，其运算结果有三个值，分别为TRUE、FALSE和NULL，其中NULL代表未知。运算优先级顺序为：NOT>AND>OR。

运算规则请参见表6-5，表中的a和b代表逻辑表达式。

表 6-5 运算规则表

a	b	a AND b的结果	a OR b的结果	NOT a的结果
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	NULL	NULL	NULL	NULL

说明

操作符AND和OR具有交换性，即交换左右两个操作数，不影响其结果。

6.9 比较操作符

比较操作符可用于所有相关的数据类型，并返回布尔类型数值。

所有比较操作符都是双目操作符，被比较的两个数据类型必须是相同的数据类型或者是可以进行隐式转换的类型。例如1<2<3这样的表达式为非法的，因为布尔值和3之间不能做比较。

GaussDB(DWS)提供的比较操作符请参见表6-6。

表 6-6 比较操作符

操作符	描述
<	小于
>	大于
<=	小于或等于
>=	大于或等于
=	等于
<> 或 !=	不等于

6.10 模式匹配操作符

数据库提供了三种实现模式匹配的方法：SQL LIKE操作符、SIMILAR TO操作符和POSIX-风格的正则表达式。除了这些基本的操作符外，还有一些函数可用于提取或替换匹配子串并在匹配位置分离一个串。

LIKE

判断字符串是否能匹配上LIKE后的模式字符串。如果字符串与提供的模式匹配，则LIKE表达式返回为真（NOT LIKE表达式返回假），否则返回为假（NOT LIKE表达式返回真）。

- 匹配规则
  a. 此操作符只有在它的模式匹配整个串的时候才能成功。如果要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。
  b. 下划线（_）代表（匹配）任何单个字符； 百分号（%）代表任意串的通配符。
  c. 要匹配文本里的下划线（_）或者百分号（%），在提供的模式里相应字符必须前导逃逸字符。逃逸字符的作用是禁用元字符的特殊含义，缺省的逃逸字符是反斜线，也可以用ESCAPE子句指定一个不同的逃逸字符。
  d. 要匹配逃逸字符本身，需写两个逃逸字符。例如要写一个包含反斜线的模式常量，那就要在SQL语句里写两个反斜线。

     说明

     参数standard_conforming_strings设置为off时，在文串常量中写的任何反斜线都需要被双写。因此写一个匹配单个反斜线的模式实际上要在语句里写四个反斜线。可通过用ESCAPE选择一个不同的逃逸字符来避免这种情况，这样反斜线就不再是LIKE的特殊字符了。但仍然是字符文本分析器的特殊字符，所以还是需要两个反斜线。也可通过写ESCAPE ''的方式不选择逃逸字符，这样可以有效地禁用逃逸机制，但是没有办法关闭下划线和百分号在模式中的特殊含义。
  e. 关键字ILIKE可以用于替换LIKE，区别是LIKE大小写敏感，ILIKE大小写不敏感。
  f. 操作符~~等效于LIKE，操作符~~*等效于ILIKE。
- 示例

```
SELECT 'abc' LIKE 'abc' AS RESULT;
 result
```



```
-----------
 t
(1 row)
SELECT 'abc' LIKE 'a%' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' LIKE '_b_' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' LIKE 'c' AS RESULT;
 result
-----------
 f
(1 row)
```

SIMILAR TO

SIMILAR TO操作符根据自己的模式判断是否匹配给定串而返回真或者假。它和LIKE非常类似，只不过它使用SQL标准定义的正则表达式理解模式。

- 匹配规则
  a. 和LIKE一样，SIMILAR TO操作符只有在它的模式匹配整个串的时候才返回真。如果要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。
  b. 下划线（_）代表（匹配）任何单个字符； 百分号（%）代表任意串的通配符。
  c. SIMILAR TO也支持下面这些从POSIX正则表达式借用的模式匹配元字符。

  表 6-7 模式匹配元字符

元字符	含义
|	表示选择（两个候选之一）。
*	表示重复前面的项零次或更多次。
+	表示重复前面的项一次或更多次。
?	表示重复前面的项零次或一次。
{m}	表示重复前面的项刚好m次。
{m,}	表示重复前面的项m次或更多次。
{m,n}	表示重复前面的项至少m次并且不超过n次。
()	把多个项组合成一个逻辑项。
[...]	声明一个字符类，就像POSIX正则表达式一样。

  d. 前导逃逸字符可以禁止所有这些元字符的特殊含义。逃逸字符的使用规则和LIKE一样。
- 注意事项

如果SIMILAR TO正则表达式重复匹配字符数量非常庞大，由于受递归大小限制，执行语句会失败并报错invalid regular expression: regular expression is too complex，可尝试调大GUC参数**max_stack_depth**。

- 正则表达式函数

  支持使用函数**substring(string from pattern for escape)**截取匹配SQL正则表达式的子字符串。

- 示例

```
SELECT 'abc' SIMILAR TO 'abc' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' SIMILAR TO 'a' AS RESULT;
 result
-----------
 f
(1 row)
SELECT 'abc' SIMILAR TO '%(b|d)%' AS RESULT;
 result
-----------
 t
(1 row)
SELECT 'abc' SIMILAR TO '(b|c)%'  AS RESULT;
 result
-----------
 f
(1 row)
```

POSIX 正则表达式

正则表达式是一个字符序列，它是定义一个串集合（一个正则集）的缩写。 如果一个串是正则表达式描述的正则集中的一员时， 那么就说这个串匹配该正则表达式。POSIX正则表达式提供了比LIKE和SIMILAR TO操作符更强大的含义。**表6-8**列出了所有可用于POSIX正则表达式模式匹配的操作符。

表 6-8 正则表达式匹配操作符

操作符	描述	例子
~	匹配正则表达式，大小写敏感	'thomas' ~ '.*thomas.*'
~*	匹配正则表达式，大小写不敏感	'thomas' ~* '.*Thomas.*'
！ ~	不匹配正则表达式，大小写敏感	'thomas' !~ '.*Thomas.*'
！ ~*	不匹配正则表达式，大小写不敏感	'thomas' !~* '.*vadim.*'

- 匹配规则
  a. 与LIKE不同，正则表达式允许匹配串里的任何位置，除非该正则表达式显式地挂接在串的开头或者结尾。
  b. 除了上文提到的元字符外， POSIX正则表达式还支持下表的模式匹配元字符。

表 6-9 模式匹配元字符

元字符	含义
^	表示串开头的匹配。
$	表示串末尾的匹配。
.	匹配任意单个字符。

- 正则表达式函数

  POSIX正则表达式支持下面函数。

  - **substring(string from pattern)**函数提供了抽取一个匹配POSIX正则表达式模式的子串的方法。
  - **regexp_replace(string, pattern, replacement [,flags])**函数提供了将匹配POSIX正则表达式模式的子串替换为新文本的功能。
  - **regexp_matches(string text, pattern text [, flags text])**函数返回一个文本数组，该数组由匹配一个POSIX正则表达式模式得到的所有被捕获子串构成。
  - **regexp_split_to_table(string text, pattern text [, flags text])**函数把一个POSIX正则表达式模式当作一个定界符来分离一个串。
  - **regexp_split_to_array(string text, pattern text [, flags text])**和regexp_split_to_table类似，是一个正则表达式分离函数，不过它的结果以一个text数组的形式返回。

    说明

    正则表达式分离函数会忽略零长度的匹配，这种匹配发生在串的开头或结尾或者正好发生在前一个匹配之后。这和正则表达式匹配的严格定义是相悖的，后者由regexp_matches实现，但是通常前者是实际中最常用的行为。

- 示例

```
 SELECT 'abc' ~ 'Abc' AS RESULT;
result
--------
 f
(1 row)
SELECT 'abc' ~* 'Abc' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'abc' !~ 'Abc' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'abc'!~* 'Abc' AS RESULT;
 result
--------
 f
(1 row)
SELECT 'abc' ~ '^a' AS RESULT;
 result
--------
 t
(1 row)
SELECT 'abc' ~ '(b|d)'AS RESULT;
 result
--------
```



```
 t
(1 row)
SELECT 'abc' ~ '^(b|c)'AS RESULT;
 result
--------
 f
(1 row)
```

虽然大部分的正则表达式搜索都能很快地执行，但是正则表达式仍可能被人为地控制，通过任意长的时间和任意量的内存进行处理。不建议从非安全模式来源接受正则表达式搜索模式，如果必须这样做，建议加上语句超时限制。使用SIMILAR TO模式的搜索具有同样的安全性危险， 因为SIMILAR TO提供了很多和POSIX-风格正则表达式相同的能力。LIKE搜索比其他两种选项简单得多，因此在接受非安全模式来源搜索时要更安全些。

6.11 聚集函数

sum(expression)

描述：所有输入行的expression总和。

返回类型：

通常情况下输入数据类型和输出数据类型是相同的，但以下情况会发生类型转换：

- 对于SMALLINT或INT输入，输出类型为BIGINT。
- 对于BIGINT输入，输出类型为NUMBER 。
- 对于浮点数输入，输出类型为DOUBLE PRECISION。

示例：

```
SELECT SUM(ss_ext_tax) FROM tpcds.STORE_SALES;
  sum
--------------
 213267594.69
(1 row)
```

max(expression)

描述：所有输入行中expression的最大值。

参数类型：任意数组、数值、字符串、日期/时间类型。

返回类型：与参数数据类型相同。

示例：

```
SELECT MAX(inv_quantity_on_hand) FROM tpcds.inventory;
   max
---------
 1000000
(1 row)
```

min(expression)

描述：所有输入行中expression的最小值。

参数类型：任意数组、数值、字符串、日期/时间类型。

返回类型：与参数数据类型相同。

示例：

```
SELECT MIN(inv_quantity_on_hand) FROM tpcds.inventory;
 min
-----
   0
(1 row)
```

avg(expression)

描述：所有输入值的均值（算术平均）。

当入参类型为DOUBLE PRECISION时，入参取值范围为1.34E-154~1.34E+154，若数值超过取值范围则报错：value out of range: overflow。如果实际使用中不可避免入参超出范围，则使用cast函数强转该列类型为numeric。

返回类型：

- 对于任何整数类型输入，结果都是NUMBER类型。
- 对于任何浮点输入，结果都是DOUBLE PRECISION类型。
- 其他，和输入数据类型相同。

示例：

```
SELECT AVG(inv_quantity_on_hand) FROM tpcds.inventory;
        avg
----------------------
 500.0387129084044604
(1 row)
```

median(expression)

描述：所有输入值的中位数值。当前只支持数值类型和interval类型。其中空值不参与计算。

返回类型：

- 对于任何整型数据输入，结果都是NUMERIC类型。否则与输入数据类型相同。
- Teradata兼容模式下，如果输入为整型，则返回的数据精度只有整数位。

示例：

```
SELECT MEDIAN(inv_quantity_on_hand) FROM tpcds.inventory;
 median
--------
    500
(1 row)
```

percentile_cont(const) within group(order by expression)

描述：返回一个对应于目标列排序中指定分位数的值，如有必要就在相邻的输入项之间插入值。其中空值不参与计算。

输入：const为在0-1之间的数值，expression当前只支持数值类型和interval类型。

返回类型：

- 对于任何整型数据输入，结果都是NUMERIC类型。否则与输入数据类型相同。
- Teradata兼容模式下，如果输入为整型，则返回的数据精度只有整数位。

示例：

```
SELECT percentile_cont(0.3) within group(order by x) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_cont
-----------------
2.2
(1 row)
SELECT percentile_cont(0.3) within group(order by x desc) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_cont
-----------------
3.8
(1 row)
```

percentile_disc(const) within group(order by expression)

描述：返回第一个在排序中位置等于或者超过指定分数的输入值。

输入：const为在0-1之间的数值，expression当前只支持数值类型和interval类型。其中空值不参与计算。

返回类型：对于任何整型数据输入，结果都是NUMERIC类型。否则，与输入数据类型相同。

示例：

```
SELECT percentile_disc(0.3) within group(order by x) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_disc
-----------------
2
(1 row)
SELECT percentile_disc(0.3) within group(order by x desc) FROM (SELECT generate_series(1,5) AS x) AS t;
percentile_disc
-----------------
4
(1 row)
```

count(expression)

描述：返回表中满足expression不为NULL的行数。

返回类型：BIGINT

示例：

```
SELECT COUNT(inv_quantity_on_hand) FROM tpcds.inventory;
  count
----------
 11158087
(1 row)
```

count(*)

描述：返回表中的记录行数。

返回类型：BIGINT

示例：

```
SELECT COUNT(*) FROM tpcds.inventory;
  count
----------
 11745000
(1 row)
```


array_agg(expression)

描述：将所有输入值（包括空）连接成一个数组。函数入参不支持数组形式。

返回类型：参数类型的数组。

示例：

创建表employeeinfo，并插入数据：

```
CREATE TABLE employeeinfo (empno smallint, ename varchar(20), job varchar(20), hiredate date,deptno
smallint);
INSERT INTO employeeinfo VALUES (7155, 'JACK', 'SALESMAN', '2018-12-01', 30);
INSERT INTO employeeinfo VALUES (7003, 'TOM', 'FINANCE', '2016-06-15', 20);
INSERT INTO employeeinfo VALUES (7357, 'MAX', 'SALESMAN', '2020-10-01', 30);

SELECT * FROM employeeinfo;
 empno | ename |   job    |      hiredate       | deptno
-------+-------+----------+---------------------+--------
  7155 | JACK  | SALESMAN | 2018-12-01 00:00:00 |     30
  7357 | MAX   | SALESMAN | 2020-10-01 00:00:00 |     30
  7003 | TOM   | FINANCE  | 2016-06-15 00:00:00 |     20
(3 rows)
```

查询部门编号为30的所有员工姓名：

```
SELECT array_agg(ename) FROM employeeinfo where deptno = 30;
 array_agg
------------
 {JACK,MAX}
(1 row)
```

查询属于同一个部门的所有员工：

```
SELECT deptno, array_agg(ename) FROM employeeinfo group by deptno;
 deptno | array_agg
--------+------------
     30 | {JACK,MAX}
     20 | {TOM}
(2 rows)

SELECT distinct array_agg(ename) OVER (PARTITION BY deptno) FROM employeeinfo;
 array_agg
------------
 {TOM}
 {JACK,MAX}
(2 rows)
```

查询所有的部门编号且去重：

```
SELECT array_agg(distinct deptno) FROM employeeinfo group by deptno;
 array_agg
-----------
 {20}
 {30}
(2 rows)
```

查询所有的部门编号去重后按降序排列：

```
SELECT array_agg(distinct deptno order by deptno desc) FROM employeeinfo;
 array_agg
-----------
 {30,20}
(1 row)
```

string_agg(expression, delimiter)

描述：将输入值连接成为一个字符串，用分隔符分开。

返回类型：和参数数据类型相同。

示例：

基于创建的表employeeinfo，查询属于同一个部门的所有员工：

```
SELECT deptno, string_agg(ename,',') FROM employeeinfo group by deptno;
 deptno | string_agg
--------+------------
     30 | JACK,MAX
     20 | TOM
(2 rows)
```

查询工号小于7156的员工：

```
SELECT string_agg(ename,',') FROM employeeinfo where empno < 7156;
 string_agg
------------
 TOM,JACK
(1 row)
```

listagg(expression [, delimiter]) WITHIN GROUP(ORDER BY order-list)

描述：将聚集列数据按WITHIN GROUP指定的排序方式排列，并用delimiter指定的分隔符拼接成一个字符串。

- expression：必选。指定聚集列名或基于列的有效表达式，不支持DISTINCT关键字和VARIADIC参数。
- delimiter：可选。指定分隔符，可以是字符串常数或基于分组列的确定性表达式，缺省时表示分隔符为空。
- order-list：必选。指定分组内的排序方式。

返回类型：text

说明

listagg是兼容Oracle 11g2的列转行聚集函数，可以指定OVER子句用作窗口函数。为了避免与函数本身WITHIN GROUP子句的ORDER BY造成二义性，listagg用作窗口函数时，OVER子句不支持ORDER BY的窗口排序或窗口框架。

示例：

聚集列是文本字符集类型：

```
SELECT deptno, listagg(ename, ',') WITHIN GROUP(ORDER BY ename) AS employees FROM emp GROUP
BY deptno;
 deptno |              employees
--------+---------------------------------------
     10 | CLARK,KING,MILLER
     20 | ADAMS,FORD,JONES,SCOTT,SMITH
     30 | ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD
(3 rows)
```

聚集列是整型：

```
SELECT deptno, listagg(mgrno, ',') WITHIN GROUP(ORDER BY mgrno NULLS FIRST) AS mgrnos FROM emp
GROUP BY deptno;
 deptno |           mgrnos
--------+-------------------------------
     10 | 7782,7839
     20 | 7566,7566,7788,7839,7902
     30 | 7698,7698,7698,7698,7698,7839
(3 rows)
```

聚集列是浮点类型：


```
SELECT job, listagg(bonus, '($); ') WITHIN GROUP(ORDER BY bonus DESC) || '($)' AS bonus FROM emp
GROUP BY job;
    job     |                      bonus
------------+-------------------------------------------------
 CLERK      | 10234.21($); 2000.80($); 1100.00($); 1000.22($)
 PRESIDENT  | 23011.88($)
 ANALYST    | 2002.12($); 1001.01($)
 MANAGER    | 10000.01($); 2399.50($); 999.10($)
 SALESMAN   | 1000.01($); 899.00($); 99.99($); 9.00($)
(5 rows)
```

聚集列是时间类型：

```
SELECT deptno, listagg(hiredate, ', ') WITHIN GROUP(ORDER BY hiredate DESC) AS hiredates FROM emp
GROUP BY deptno;
 deptno |                                                hiredates
--------
+-----------------------------------------------------------------------------------------------------------------------------
--
     10 | 1982-01-23 00:00:00, 1981-11-17 00:00:00, 1981-06-09 00:00:00
     20 | 2001-04-02 00:00:00, 1999-12-17 00:00:00, 1987-05-23 00:00:00, 1987-04-19 00:00:00, 1981-12-03
00:00:00
     30 | 2015-02-20 00:00:00, 2010-02-22 00:00:00, 1997-09-28 00:00:00, 1981-12-03 00:00:00, 1981-09-08
00:00:00, 1981-05-01 00:00:00
(3 rows)
```

聚集列是时间间隔类型：

```
SELECT deptno, listagg(vacationTime, '; ') WITHIN GROUP(ORDER BY vacationTime DESC) AS vacationTime
FROM emp GROUP BY deptno;
 deptno |                                vacationtime
--------+-------------------------------------------------------------------------------
     10 | 1 year 30 days; 40 days; 10 days
     20 | 70 days; 36 days; 9 days; 5 days
     30 | 1 year 1 mon; 2 mons 10 days; 30 days; 12 days 12:00:00; 4 days 06:00:00; 24:00:00
(3 rows)
```

分隔符缺省时，默认为空：

```
SELECT deptno, listagg(job) WITHIN GROUP(ORDER BY job) AS jobs FROM emp GROUP BY deptno;
 deptno |                    jobs
--------+---------------------------------------------
     10 | CLERKMANAGERPRESIDENT
     20 | ANALYSTANALYSTCLERKCLERKMANAGER
     30 | CLERKMANAGERSALESMANSALESMANSALESMANSALESMAN
(3 rows)
```

listagg作为窗口函数时，OVER子句不支持ORDER BY的窗口排序，listagg列为对应分组的有序聚集：

```
SELECT deptno, mgrno, bonus, listagg(ename,'; ') WITHIN GROUP(ORDER BY hiredate) OVER(PARTITION
BY deptno) AS employees FROM emp;
 deptno | mgrno |  bonus   |                 employees
--------+-------+----------+-------------------------------------------
     10 |  7839 | 10000.01 | CLARK; KING; MILLER
     10 |       | 23011.88 | CLARK; KING; MILLER
     10 |  7782 | 10234.21 | CLARK; KING; MILLER
     20 |  7566 |  2002.12 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7566 |  1001.01 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7788 |  1100.00 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7902 |  2000.80 | FORD; SCOTT; ADAMS; SMITH; JONES
     20 |  7839 |   999.10 | FORD; SCOTT; ADAMS; SMITH; JONES
     30 |  7839 |  2399.50 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |     9.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |  1000.22 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |    99.99 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |  1000.01 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
     30 |  7698 |   899.00 | BLAKE; TURNER; JAMES; MARTIN; WARD; ALLEN
(14 rows)
```


group_concat(expression [ORDER BY {col_name | expr} [ASC | DESC]] [SEPARATOR str_val])

描述：将列数据使用指定的str_val分隔符，按照ORDER BY子句指定的排序方式拼接成字符串，ORDER BY子句必须指定排序方式，不支持ORDER BY 1的写法。

- expression：必选，指定列名或基于列的有效表达式，不支持DISTINCT关键字和VARIADIC参数。
- str_val：可选，指定的分隔符，可以是字符串常数或基于分组列的确定性表达式。缺省时表示分隔符为逗号。

返回类型：text

说明

group_concat函数仅8.1.2及以上版本支持。

示例：

默认分隔符为逗号：

```
SELECT group_concat(sname) FROM group_concat_test;
              group_concat
-------------------------------------------
 ADAMS,FORD,JONES,KING,MILLER,SCOTT,SMITH
(1 row)
```

group_concat函数支持自定义分隔符：

```
SELECT group_concat(sname separator ';') from group_concat_test;
              group_concat
-------------------------------------------
 ADAMS;FORD;JONES;KING;MILLER;SCOTT;SMITH
(1 row)
```

group_concat函数支持ORDER BY子句，将列数据进行有序拼接：

```
SELECT group_concat(sname order by snumber separator ';') FROM group_concat_test;
              group_concat
-------------------------------------------
 MILLER;FORD;SCOTT;SMITH;KING;JONES;ADAMS
(1 row)
```

covar_pop(Y, X)

描述：总体协方差。

返回类型：double precision

示例：

```
 SELECT COVAR_POP(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
    covar_pop
------------------
 829.749627587403
(1 row)
```

covar_samp(Y, X)

描述：样本协方差。

返回类型：double precision

示例：

```
SELECT COVAR_SAMP(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
    covar_samp
------------------
 830.052235037289
(1 row)
```

stddev_pop(expression)

描述：总体标准差。

当入参类型为DOUBLE PRECISION时，入参取值范围为1.34E-154~1.34E+154，若数值超过取值范围则报错：value out of range: overflow。如果实际使用中不可避免入参超出范围，则使用cast函数强转该列类型为numeric。

返回类型：对于浮点类型的输入返回double precision，其他输入返回numeric。

示例：

```
SELECT STDDEV_POP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
    stddev_pop
------------------
 289.224294957556
(1 row)
```

stddev_samp(expression)

描述：样本标准差。

当入参类型为DOUBLE PRECISION时，入参取值范围为1.34E-154~1.34E+154，若数值超过取值范围则报错：value out of range: overflow。如果实际使用中不可避免入参超出范围，则使用cast函数强转该列类型为numeric。

返回类型：对于浮点类型的输入返回double precision，其他输入返回numeric。

示例：

```
SELECT STDDEV_SAMP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
   stddev_samp
------------------
 289.224359757315
(1 row)
```

var_pop(expression)

描述：总体方差（总体标准差的平方）。

当入参类型为DOUBLE PRECISION时，入参取值范围为1.34E-154~1.34E+154，若数值超过取值范围则报错：value out of range: overflow。如果实际使用中不可避免入参超出范围，则使用cast函数强转该列类型为numeric。

返回类型：对于浮点类型的输入返回double precision类型，其他输入返回numeric类型。

示例：

```
SELECT VAR_POP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
      var_pop
--------------------
 83650.692793695475
(1 row)
```


var_samp(expression)

描述：样本方差（样本标准差的平方）。

当入参类型为DOUBLE PRECISION时，入参取值范围为1.34E-154~1.34E+154，若数值超过取值范围则报错：value out of range: overflow。如果实际使用中不可避免入参超出范围，则使用cast函数强转该列类型为numeric。

返回类型：对于浮点类型的输入返回double precision类型，其他输入返回numeric类型。

示例：

```
SELECT VAR_SAMP(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
     var_samp
--------------------
 83650.730277028768
(1 row)
```

bit_and(expression)

描述：所有非NULL输入值的按位与（AND），如果全部输入值皆为NULL，那么结果也为NULL 。

返回类型：和参数数据类型相同。

示例：

```
SELECT BIT_AND(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
 bit_and
---------
       0
(1 row)
```

bit_or(expression)

描述：所有非NULL输入值的按位或（OR），如果全部输入值皆为NULL，那么结果也为NULL。

返回类型：和参数数据类型相同

示例：

```
SELECT BIT_OR(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
 bit_or
--------
   1023
(1 row)
```

bool_and(expression)

描述：如果所有输入值都是真，则为真，否则为假。

返回类型：bool

示例：

```
SELECT bool_and(100 <2500);
 bool_and
----------
 t
(1 row)
```


bool_or(expression)

描述：如果所有输入值只要有一个为真，则为真，否则为假。

返回类型：bool

示例：

```
SELECT bool_or(100 <2500);
 bool_or
----------
 t
(1 row)
```

corr(Y, X)

描述：相关系数。

返回类型：double precision

示例：

```
SELECT CORR(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
       corr
--------------------
 .0381383624904186
(1 row)
```

every(expression)

描述：等效于bool_and。

返回类型：bool

示例：

```
SELECT every(100 <2500);
 every
-------
 t
(1 row)
```

rank(expression)

描述：根据expression对不同组内的元组进行跳跃排序。

返回类型：bigint

示例：

```
SELECT d_moy, d_fy_week_seq, rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM
tpcds.date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;
   d_moy | d_fy_week_seq | rank
--------+----------------+------
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              1 |    1
     1 |              2 |    8
     1 |              2 |    8
     1 |              2 |    8
     1 |              2 |    8
```

```
   1 |          2 |   8
   1 |          2 |   8
   1 |          2 |   8
   1 |          3 |  15
   1 |          3 |  15
   1 |          3 |  15
   1 |          3 |  15
   1 |          3 |  15
   1 |          3 |  15
   1 |          3 |  15
   1 |          3 |  15
   1 |          4 |  22
   1 |          4 |  22
   1 |          4 |  22
   1 |          4 |  22
   1 |          4 |  22
   1 |          4 |  22
   1 |          4 |  22
   1 |          5 |  29
   1 |          5 |  29
   2 |          5 |   1
   2 |          5 |   1
   2 |          5 |   1
   2 |          5 |   1
   2 |          5 |   1
   2 |          6 |   6
   2 |          6 |   6
   2 |          6 |   6
   2 |          6 |   6
   2 |          6 |   6
   2 |          6 |   6
   2 |          6 |   6
(42 rows)
```

regr_avgx(Y, X)

描述：自变量的平均值 (sum(X)/N)。

返回类型：double precision

示例：

```
SELECT REGR_AVGX(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
    regr_avgx
------------------
 578.606576740795
(1 row)
```

regr_avgy(Y, X)

描述：因变量的平均值 (sum(Y)/N)。

返回类型：double precision

示例：

```
SELECT REGR_AVGY(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
    regr_avgy
------------------
 50.0136711629602
(1 row)
```

regr_count(Y, X)

描述：两个表达式都不为NULL的输入行数。

返回类型：bigint

示例：

```
SELECT REGR_COUNT(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
 regr_count
------------
       2743
(1 row)
```

regr_intercept(Y, X)

描述：根据所有输入的点(X, Y)按照最小二乘法拟合成一个线性方程，然后返回该直线的Y轴截距。

返回类型：double precision

示例：

```
SELECT REGR_INTERCEPT(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
  regr_intercept
------------------
 49.2040847848607
(1 row)
```

regr_r2(Y, X)

描述：相关系数的平方。

返回类型：double precision

示例：

```
SELECT REGR_R2(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
      regr_r2
--------------------
 .00145453469345058
(1 row)
```

regr_slope(Y, X)

描述：根据所有输入的点(X, Y)按照最小二乘法拟合成一个线性方程， 然后返回该直线的斜率。

返回类型：double precision

示例：

```
SELECT REGR_SLOPE(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
     regr_slope
--------------------
 .00139920009665259
(1 row)
```

regr_sxx(Y, X)

描述：sum(X^2) - sum(X)^2/N （自变量的“平方和”）。

返回类型：double precision

示例：

```
SELECT REGR_SXX(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
     regr_sxx
------------------
```



```
 1626645991.46135
(1 row)
```

regr_sxy(Y, X)

描述：sum(X*Y) - sum(X) * sum(Y)/N （自变量和因变量的“乘方积”）。

返回类型：double precision

示例：

```
SELECT REGR_SXY(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
    regr_sxy
------------------
 2276003.22847225
(1 row)
```

regr_syy(Y, X)

描述：sum(Y^2) - sum(Y)^2/N（因变量的“平方和”）。

返回类型：double precision

示例：

```
SELECT REGR_SYY(sr_fee, sr_net_loss) FROM tpcds.store_returns WHERE sr_customer_sk < 1000;
   regr_syy
-----------------
 2189417.6547314
(1 row)
```

stddev(expression)

描述：stddev_samp的别名。

当入参类型为DOUBLE PRECISION时，入参取值范围为1.34E-154~1.34E+154，若数值超过取值范围则报错：value out of range: overflow。如果实际使用中不可避免入参超出范围，则使用cast函数强转该列类型为numeric。

返回类型：对于浮点类型的输入返回double precision，其他输入返回numeric。

示例：

```
SELECT STDDEV(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
     stddev
------------------
 289.224359757315
(1 row)
```

variance(expexpression,ression)

描述：var_samp的别名。

当入参类型为DOUBLE PRECISION时，入参取值范围为1.34E-154~1.34E+154，若数值超过取值范围则报错：value out of range: overflow。如果实际使用中不可避免入参超出范围，则使用cast函数强转该列类型为numeric。

返回类型：对于浮点类型的输入返回double precision类型，其他输入返回numeric类型。

示例：


```
SELECT VARIANCE(inv_quantity_on_hand) FROM tpcds.inventory WHERE inv_warehouse_sk = 1;
     variance
--------------------
 83650.730277028768
(1 row)
```

checksum(expression)

描述：返回所有输入值的CHECKSUM值。使用该函数可以用来验证GaussDB(DWS)数据库（不支持GaussDB(DWS)之外的其他数据库）的备份恢复或者数据迁移操作前后表中的数据是否相同。在备份恢复或者数据迁移操作前后都需要用户通过手工执行SQL命令的方式获取执行结果，通过对比获取的执行结果判断操作前后表中的数据是否相同。

说明

- 对于大表，CHECKSUM函数可能会需要很长时间。
- 如果某两表的CHECKSUM值不同，则表明两表的内容是不同的。由于CHECKSUM函数中使用散列函数不能保证无冲突，因此两个不同内容的表可能会得到相同的CHECKSUM值，存在这种情况的可能性较小。对于列进行的CHECKSUM也存在相同的情况。
- 对于时间类型timestamp, timestamptz和smalldatetime，计算CHECKSUM值时请确保时区设置一致。

- 若计算某列的CHECKSUM值，且该列类型可以默认转为TEXT类型，则expression为列名。
- 若计算某列的CHECKSUM值，且该列类型不能默认转为TEXT类型，则expression为列名::TEXT。
- 若计算所有列的CHECKSUM值，则expression为表名::TEXT。

可以默认转换为TEXT类型的类型包括：char, name, int8, int2, int1, int4, raw, pg_node_tree, float4, float8, bpchar, varchar, nvarchar2, date, timestamp, timestamptz, numeric, smalldatetime，其他类型需要强制转换为TEXT。

返回类型：numeric

示例：

表中可以默认转为TEXT类型的某列的CHECKSUM值：

```
SELECT CHECKSUM(inv_quantity_on_hand) FROM tpcds.inventory;
    checksum
-------------------
 24417258945265247
(1 row)
```

表中不能默认转为TEXT类型的某列的CHECKSUM值（注意此时CHECKSUM参数是列名::TEXT）：

```
SELECT CHECKSUM(inv_quantity_on_hand::TEXT) FROM tpcds.inventory;
    checksum
-------------------
 24417258945265247
(1 row)
```

表中所有列的CHECKSUM值。注意此时CHECKSUM参数是表名::TEXT，且表名前不加Schema：

```
SELECT CHECKSUM(inventory::TEXT) FROM tpcds.inventory;
    checksum
-------------------
 25223696246875800
(1 row)
```


6.12 窗口函数

普通的聚集函数只能用来计算一行内的结果，或者把所有行聚集成一行结果。而窗口函数可以跨行计算，并且把结果填到每一行中。

通过查询筛选出的行的某些部分，窗口调用函数实现了类似于**聚集函数**的功能，所以聚集函数也可以作为窗口函数使用。 窗口函数可以扫描所有的行，并同时将原始数据和聚集分析结果同时显示出来。

注意事项

- 列存表目前只支持窗口函数rank(expression)和row_number(expression)，以及聚集函数的sum，count，avg，min和max，而行存表没有限制。
- 单个查询中可以包含一个或多个窗口函数表达式。
- 窗口函数仅能出现在输出列中。如果需要使用窗口函数的值进行条件过滤，需要将窗口函数嵌套在子查询中，在外层使用窗口函数表达式的别名进行条件过滤。例如：
  ```
  SELECT classid, id, score FROM(SELECT *, avg(score) OVER(PARTITION BY classid) as avg_score FROM score) WHERE score >= avg_score;
  ```
- 窗口函数所在查询块中支持使用GROUP BY表达式进行分组去重，但要求窗口函数中的PARTITION BY子句中必须是GROUP BY表达式的子集，以保证窗口函数在GROUP BY列去重后的结果上进行窗口运算，同时ORDER BY子句的表达式也需要是GROUP BY表达式的子集，或聚集运算的聚集函数。例如：
  ```
  SELECT classid,rank() OVER(PARTITION BY classid ORDER BY sum(score)) as avg_score FROM score GROUP BY classid, id;
  ```

语法格式

窗口函数需要特殊的关键字OVER语句来指定窗口触发窗口函数。OVER语句用于对数据进行分组，并对组内元素进行排序。窗口函数用于给组内的值生成序号：

```
function_name ([expression [, expression ... ]]) OVER ( window_definition )
function_name ([expression [, expression ... ]]) OVER window_name
function_name ( * ) OVER ( window_definition )
function_name ( * ) OVER window_name
```

其中window_definition子句option为：

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

PARTITION BY选项指定了将具有相同PARTITION BY表达式值的行分为一组。

ORDER BY选项用于控制窗口函数处理行的顺序。ORDER BY后面必须跟字段名，若ORDER BY后面跟数字，该数字会被按照常量处理，对目标列没有起到排序的作用。

frame_clause子句option为：

```
[ RANGE | ROWS ] frame_start
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

当需要指定一个窗口对分组内所有行结果进行计算时，我们需要指定窗口区间开始的行和结束的行。窗口区间支持RANGE、ROWS两种模式，ROWS以物理单位（行）指定窗口，RANGE将窗口指定为逻辑偏移量。

RANGE、ROWS中可以使用BETWEEN frame_start AND frame_end指定边界可取值。如果仅指定frame_start，则frame_end默认为CURRENT ROW。

frame_start和frame_end取值为：

- CURRENT ROW，当前行。
- N PRECEDING，当前行向前第n行。
- UNBOUNDED PRECEDING，当前PARTITION的第1行。
- N FOLLOWING，当前行向后第n行。
- UNBOUNDED FOLLOWING，当前PARTITION的最后1行。

需要注意，frame_start不能为UNBOUNDED FOLLOWING，frame_end不能为UNBOUNDED PRECEDING，并且frame_end选项不能比上面取值中出现的frame_start选项早。例如RANGE BETWEEN CURRENT ROW AND N PRECEDING是不被允许的。

RANK()

描述：RANK函数为各组内值生成跳跃排序序号，其中相同的值具有相同序号，但相同值占用多个编号。

返回值类型：BIGINT

示例：

给定表score(id, classid, score)，每行表示学生id，所在班级id以及考试成绩。

使用RANK函数对学生成绩进行排序：

```
CREATE TABLE score(id int,classid int,score int);
INSERT INTO score VALUES(1,1,95),(2,2,95),(3,2,85),(4,1,70),(5,2,88),(6,1,70);

SELECT id, classid, score,RANK() OVER(ORDER BY score DESC) FROM score;
 id | classid | score | rank
----+---------+-------+------
  1 |       1 |    95 |    1
  2 |       2 |    95 |    1
  6 |       1 |    70 |    5
  4 |       1 |    70 |    5
  5 |       2 |    88 |    3
  3 |       2 |    85 |    4
(6 rows)
```

ROW_NUMBER()

描述：ROW_NUMBER函数为各组内值生成连续排序序号，其中相同的值其序号也不相同。

返回值类型：BIGINT

示例：

```
SELECT id, classid, score,ROW_NUMBER() OVER(ORDER BY score DESC) FROM score ORDER BY score DESC;
 id | classid | score | row_number
----+---------+-------+------------
  1 |       1 |    95 |          1
  2 |       2 |    95 |          2
  5 |       2 |    88 |          3
  3 |       2 |    85 |          4
  6 |       1 |    70 |          5
```

```
 4 |      1 |   70 |         6
(6 rows)
```

DENSE_RANK()

描述：DENSE_RANK函数为各组内值生成连续排序序号，其中相同的值具有相同序号，相同值只占用一个编号。

返回值类型：BIGINT

示例：

```
SELECT id, classid, score,DENSE_RANK() OVER(ORDER BY score DESC) FROM score;
 id | classid | score | dense_rank
----+---------+-------+------------
  1 |       1 |    95 |          1
  2 |       2 |    95 |          1
  5 |       2 |    88 |          2
  3 |       2 |    85 |          3
  6 |       1 |    70 |          4
  4 |       1 |    70 |          4
(6 rows)
```

PERCENT_RANK()

描述：PERCENT_RANK函数为各组内对应值生成相对序号，即根据公式 (rank - 1) / (total rows - 1)计算所得的值。其中rank为该值依据RANK函数所生成的对应序号，totalrows为该分组内的总元素个数。

返回值类型：DOUBLE PRECISION

示例：

```
SELECT id, classid, score,PERCENT_RANK() OVER(ORDER BY score DESC) FROM score;
 id | classid | score | percent_rank
----+---------+-------+--------------
  1 |       1 |    95 |            0
  2 |       2 |    95 |            0
  3 |       2 |    85 |           .6
  4 |       1 |    70 |           .8
  5 |       2 |    88 |           .4
  6 |       1 |    70 |           .8
(6 rows)
```

CUME_DIST()

描述：CUME_DIST函数为各组内对应值生成累积分布序号。即根据公式(小于等于当前值的数据行数)/(该分组总行数totalrows)计算所得的相对序号。

返回值类型：DOUBLE PRECISION

示例：

```
SELECT id,classid,score,CUME_DIST() OVER(ORDER BY score DESC) FROM score;
 id | classid | score |     cume_dist
----+---------+-------+-------------------
  1 |       1 |    95 | .333333333333333
  2 |       2 |    95 | .333333333333333
  5 |       2 |    88 |               .5
  3 |       2 |    85 | .666666666666667
  4 |       1 |    70 |                1
  6 |       1 |    70 |                1
(6 rows)
```

NTILE(num_buckets integer)

描述：NTILE函数根据num_buckets integer将有序的数据集合平均分配到num_buckets所指定数量的桶中，并将桶号分配给每一行。分配时应尽量做到平均分配。

返回值类型：INTEGER

示例：

```
SELECT id,classid,score,NTILE(3) OVER(ORDER BY score DESC) FROM score;
 id | classid | score | ntile
----+---------+-------+-------
  1 |       1 |    95 |     1
  2 |       2 |    95 |     1
  5 |       2 |    88 |     2
  3 |       2 |    85 |     2
  4 |       1 |    70 |     3
  6 |       1 |    70 |     3
(6 rows)
```

LAG(value any [, offset integer [, default any]])

描述：LAG函数为各组内对应值生成滞后值。即当前值对应的行数往前偏移offset位后所得行的value值作为序号。若经过偏移后行数不存在，则对应结果取为default值。若无指定，在默认情况下，offset取为1，default值取为NULL。

返回值类型：与参数数据类型相同。

示例：

```
SELECT id,classid,score,LAG(id,3) OVER(ORDER BY score DESC) FROM score;
 id | classid | score | lag
----+---------+-------+-----
  1 |       1 |    95 |
  2 |       2 |    95 |
  5 |       2 |    88 |
  3 |       2 |    85 |   1
  4 |       1 |    70 |   2
  6 |       1 |    70 |   5
(6 rows)
```

LEAD(value any [, offset integer [, default any]])

描述：LEAD函数为各组内对应值生成提前值。即当前值对应的行数向后偏移offset位后所得行的value值作为序号。若经过向后偏移后行数超过当前组内的总行数，则对应结果取为default值。若无指定，在默认情况下，offset取为1，default值取为NULL。

返回值类型：与参数数据类型相同。

示例：

```
SELECT id,classid,score,LEAD(id,3) OVER(ORDER BY score DESC) FROM score;
 id | classid | score | lead
----+---------+-------+------
  1 |       1 |    95 |    3
  2 |       2 |    95 |    4
  5 |       2 |    88 |    6
  3 |       2 |    85 |
  4 |       1 |    70 |
  6 |       1 |    70 |
(6 rows)
```

FIRST_VALUE(value any)

描述：FIRST_VALUE函数取各组内的第一个值作为返回结果。

返回值类型：与参数数据类型相同。

示例：

```
SELECT id,classid,score,FIRST_VALUE(id) OVER(ORDER BY score DESC) FROM score;
 id | classid | score | first_value
----+---------+-------+-------------
  1 |       1 |    95 |           1
  2 |       2 |    95 |           1
  5 |       2 |    88 |           1
  3 |       2 |    85 |           1
  4 |       1 |    70 |           1
  6 |       1 |    70 |           1
(6 rows)
```

LAST_VALUE(value any)

描述：LAST_VALUE函数取各组内的最后一个值作为返回结果。

返回值类型：与参数数据类型相同。

示例：

```
SELECT id,classid,score,LAST_VALUE(id) OVER(ORDER BY score DESC) FROM score;
 id | classid | score | last_value
----+---------+-------+------------
  1 |       1 |    95 |          2
  2 |       2 |    95 |          2
  5 |       2 |    88 |          5
  3 |       2 |    85 |          3
  4 |       1 |    70 |          6
  6 |       1 |    70 |          6
(6 rows)
```

NTH_VALUE(value any, nth integer)

描述：NTH_VALUE函数返回该组内的第nth行作为结果。若该行不存在，则默认返回NULL。

返回值类型：与参数数据类型相同。

示例：

```
SELECT id,classid,score,NTH_VALUE(id,3) OVER(ORDER BY score DESC) FROM score;
 id | classid | score | nth_value
----+---------+-------+-----------
  1 |       1 |    95 |
  2 |       2 |    95 |
  5 |       2 |    88 |         5
  3 |       2 |    85 |         5
  4 |       1 |    70 |         5
  6 |       1 |    70 |         5
(6 rows)
```

6.13 类型转换函数

cast(x as y)

描述：类型转换函数，将x转换成y指定的类型。

示例：

```
SELECT cast('22-oct-1997' as timestamp);
     timestamp
---------------------
 1997-10-22 00:00:00
(1 row)
```

hextoraw(string)

描述：将CHAR、VARCHAR2、NCHAR或NVARCHAR2数据类型中包含十六进制数字的字符转换为**RAW数据类型**。

返回值类型：raw

示例：

```
SELECT hextoraw('7D');
 hextoraw
----------
 7D
(1 row)
```

numtoday(numeric)

描述：将数字类型的值转换为指定格式的时间戳。

返回值类型：timestamp

示例：

```
SELECT numtoday(2);
 numtoday
----------
 2 days
(1 row)
```

pg_systimestamp()

描述：获取系统时间戳。

返回值类型：timestamp with time zone

示例：

```
SELECT pg_systimestamp();
       pg_systimestamp
-------------------------------
 2015-10-14 11:21:28.317367+08
(1 row)
```

rawtohex(string)

描述：将**RAW数据类型**值转换为相应的十六进制表示的字符串。string中的每个字节都被转换一个双字节的字符串。

结果为输入字符的ACSII码，以十六进制表示。

返回值类型：varchar

示例：

```
SELECT rawtohex('1234567');
   rawtohex
```



```
----------------
 31323334353637
(1 row)
```

to_char (datetime/interval [, fmt])

描述：将一个DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE或者TIMESTAMP WITH LOCAL TIME ZONE类型的DATETIME或者INTERVAL值按照fmt指定的格式转换为VARCHAR类型。

- 可选参数fmt可以为以下几类：日期、时间、星期、季度和世纪。每类都可以有不同的模板，模板之间可以合理组合，常见的模板有：HH、MM、SS、YYYY、MM、DD。
- 模板可以有修饰词，常用的修饰词是FM，可以用来抑制前导的零或尾随的空白。

返回值类型：varchar

示例：

```
SELECT to_char(current_timestamp,'HH12:MI:SS');
 to_char
----------
 10:19:26
(1 row)
SELECT to_char(current_timestamp,'FMHH12:FMMI:FMSS');
 to_char
----------
 10:19:46
(1 row)
```

to_char(double precision, text)

描述：将双精度类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(125.8::real, '999D99');
 to_char
---------
  125.80
(1 row)
```

to_char (integer/number[, fmt])

描述：将一个整型或者浮点类型的值转换为指定格式的字符串。

- 可选参数fmt可以为以下几类：十进制字符、“分组”符、正负号和货币符号，每类都可以有不同的模板，模板之间可以合理组合，常见的模板有：9、0、,（千分隔符）、.（小数点）。
- 模板可以有类似FM的修饰词，但FM不抑制由模板0指定而输出的0。
- 要将整型类型的值转换成对应16进制值的字符串，使用模板X或x。

返回值类型：varchar

示例：

```
SELECT to_char(1485,'9,999');
 to_char
---------
```



```
 1,485
(1 row)
SELECT to_char( 1148.5,'9,999.999');
 to_char
-----------
 1,148.500
(1 row)
SELECT to_char(148.5,'990999.909');
  to_char
-------------
   0148.500
(1 row)
SELECT to_char(123,'XXX');
 to_char
---------
  7B
(1 row)
```

to_char(interval, text)

描述：将时间间隔类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(interval '15h 2m 12s', 'HH24:MI:SS');
 to_char
----------
 15:02:12
(1 row)
```

to_char(int, text)

描述：将整数类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(125, '999');
 to_char
---------
 125
(1 row)
```

to_char(numeric, text)

描述：将数字类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(-125.8, '999D99S');
 to_char
---------
 125.80-
(1 row)
```

to_char (string)

描述：将CHAR、VARCHAR、VARCHAR2、CLOB类型转换为VARCHAR类型。

如使用该函数对CLOB类型进行转换，且待转换CLOB类型的值超出目标类型的范围，则返回错误。

返回值类型：varchar

示例：

```
SELECT to_char('01110');
 to_char
---------
 01110
(1 row)
```

to_char(timestamp, text)

描述：将时间戳类型的值转换为指定格式的字符串。

返回值类型：text

示例：

```
SELECT to_char(current_timestamp, 'HH12:MI:SS');
 to_char
----------
 10:55:59
(1 row)
```

to_clob(char/nchar/varchar/nvarchar/varchar2/nvarchar2/text/raw)

描述：将RAW类型或者文本字符集类型CHAR、NCHAR、VARCHAR、VARCHAR2、NVARCHAR2、TEXT转成CLOB类型。

返回值类型：clob

示例：

```
SELECT to_clob('ABCDEF'::RAW(10));
 to_clob
---------
 ABCDEF
(1 row)
SELECT to_clob('hello111'::CHAR(15));
 to_clob
----------
 hello111
(1 row)
SELECT to_clob('gauss123'::NCHAR(10));
 to_clob
----------
 gauss123
(1 row)
SELECT to_clob('gauss234'::VARCHAR(10));
 to_clob
----------
 gauss234
(1 row)
SELECT to_clob('gauss345'::VARCHAR2(10));
 to_clob
----------
 gauss345
(1 row)
SELECT to_clob('gauss456'::NVARCHAR2(10));
 to_clob
----------
 gauss456
(1 row)
```



```
SELECT to_clob('World222!'::TEXT);
  to_clob
-----------
 World222!
(1 row)
```

to_date(text)

描述：将文本类型的值转换为指定格式的时间戳。

返回值类型：timestamp

示例：

```
SELECT to_date('2015-08-14');
       to_date
---------------------
 2015-08-14 00:00:00
(1 row)
```

to_date(text, text)

描述：将字符串类型的值转换为指定格式的日期。

返回值类型：timestamp

示例：

```
SELECT to_date('05 Dec 2000', 'DD Mon YYYY');
       to_date
---------------------
 2000-12-05 00:00:00
(1 row)
```

to_date(string, fmt)

描述：将字符串string按fmt指定格式转化为DATE类型的值。该函数不能直接支持CLOB类型，但是CLOB类型的参数能够通过隐式转换实现。

返回值类型：date

示例：

```
SELECT TO_DATE('05 Dec 2010','DD Mon YYYY');
       to_date
---------------------
 2010-12-05 00:00:00
(1 row)
```

to_number (expr [, fmt])

描述：将expr按指定格式转换为一个NUMBER类型的值。

类型转换格式请参考**表6-10**。

转换十六进制字符串为十进制数字时，最多支持16个字节的十六进制字符串转换为无符号数。

转换十六进制字符串为十进制数字时，格式字符串中不允许出现除“x”或“X”以外的其他字符，否则报错。

返回值类型：number

示例：

```
SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
  -12454.8
(1 row)
```

to_number(text, text)

描述：将字符串类型的值转换为指定格式的数字。

返回值类型：numeric

示例：

```
SELECT to_number('12,454.8-', '99G999D9S');
 to_number
-----------
  -12454.8
(1 row)
```

to_timestamp(double precision)

描述：把UNIX纪元转换成时间戳。

返回值类型：timestamp with time zone

示例：

```
SELECT to_timestamp(1284352323);
      to_timestamp
------------------------
 2010-09-13 12:32:03+08
(1 row)
```

to_timestamp(string [,fmt])

描述：将字符串string按fmt指定的格式转换成时间戳类型的值。不指定fmt时，按参数nls_timestamp_format所指定的格式转换。

GaussDB(DWS)的to_timestamp中，

- 如果输入的年份YYYY=0，系统报错。
- 如果输入的年份YYYY<0，在fmt中指定SYYYY，则正确输出公元前绝对值n的年份。

fmt中出现的字符必须与日期/时间格式化的模式相匹配，否则报错。

返回值类型：timestamp without time zone

示例：

```
SHOW nls_timestamp_format;
    nls_timestamp_format
-----------------------------
 DD-Mon-YYYY HH:MI:SS.FF AM
(1 row)

SELECT to_timestamp('12-sep-2014');
    to_timestamp
---------------------
 2014-09-12 00:00:00
(1 row)
```

```
SELECT to_timestamp('12-Sep-10 14:10:10.123000','DD-Mon-YY HH24:MI:SS.FF');
      to_timestamp
-------------------------
 2010-09-12 14:10:10.123
(1 row)
SELECT to_timestamp('-1','SYYYY');
      to_timestamp
------------------------
 0001-01-01 00:00:00 BC
(1 row)
SELECT to_timestamp('98','RR');
    to_timestamp
---------------------
 1998-01-01 00:00:00
(1 row)
SELECT to_timestamp('01','RR');
    to_timestamp
---------------------
 2001-01-01 00:00:00
(1 row)
```

to_timestamp(text, text)

描述：将字符串类型的值转换为指定格式的时间戳。

返回值类型：timestamp

示例：

```
SELECT to_timestamp('05 Dec 2000', 'DD Mon YYYY');
    to_timestamp
---------------------
 2000-12-05 00:00:00
(1 row)
```

表6-10显示了可以用于格式化数值的模板模式，适用于函数to_number。

表 6-10 数值格式化的模版模式

模式	描述
9	带有指定数值位数的值
0	带前导零的值
.（句点）	小数点
,（逗号）	分组（千）分隔符
PR	尖括号内负值
S	带符号的数值（使用区域设置）
L	货币符号（使用区域设置）
D	小数点（使用区域设置）
G	分组分隔符（使用区域设置）
MI	在指明的位置的负号（如果数字 < 0）
PL	在指明的位置的正号（如果数字 > 0）
SG	在指明的位置的正/负号

模式	描述
RN	罗马数字（输入在 1 和 3999 之间）
TH或th	序数后缀
V	移动指定位（小数）

表6-11显示了可以用于格式化日期和时间值的模版，这些模式适用于函数to_date、to_timestamp、to_char和参数nls_timestamp_format。

表 6-11 用于日期/时间格式化的模式

类别	模式	描述
小时	HH	一天的小时数（01-12）
	HH12	一天的小时数（01-12）
	HH24	一天的小时数（00-23）
分钟	MI	分钟（00-59）
秒	SS	秒（00-59）
	FF	微秒（000000-999999）
	SSSSS	午夜后的秒（0-86399）
上、下午	AM或A.M.	上午标识
	PM或P.M.	下午标识
年	Y,YYY	带逗号的年（4和更多位）
	SYYYY	公元前四位年
	YYYY	年（4和更多位）
	YYY	年的后三位
	YY	年的后两位
	Y	年的最后一位
	IYYY	ISO年（4位或更多位）
	IYY	ISO年的最后三位
	IY	ISO年的最后两位
	I	ISO年的最后一位

类别	模式	描述
	RR	年的后两位（可在21世纪存储20世纪的年份） 规则如下： ● 输入的两位年份在00~49之间： 当前年份的后两位在00~49之间，返回值年份的前两位和当前年份的前两位相同； 当前年份的后两位在50~99之间，返回值年份的前两位是当前年份的前两位加1。 ● 输入的两位年份在50~99之间： 当前年份的后两位在00~49之间，返回值年份的前两位是当前年份的前两位减1； 当前年份的后两位在50~99之间，返回值年份的前两位和当前年份的前两位相同。
	RRRR	可接收4位年或两位年。若是两位，则和RR的返回值相同，若是四位，则和YYYY相同。
	● BC或B.C. ● AD或A.D.	纪元标识。BC（公元前），AD（公元后）。
月	MONTH	全长大写月份名（空白填充为9字符）
	MON	大写缩写月份名（3字符）
	MM	月份数（01-12）
	RM	罗马数字的月份（I-XII； I=JAN）（大写）
天	DAY	全长大写日期名（空白填充为9字符）
	DY	缩写大写日期名（3字符）
	DDD	一年里的日（001-366）
	DD	一个月里的日（01-31）
	D	一周里的日（1-7； 周日是 1）
周	W	一个月里的周数（1-5）（第一周从该月第一天开始）
	WW	一年里的周数（1-53）（第一周从该年的第一天开始）
	IW	ISO一年里的周数（第一个星期四在第一周里）
世纪	CC	世纪（2位）（21 世纪从 2001-01-01 开始）
儒略日	J	儒略日（自公元前 4712 年 1 月 1 日来的天数）
季度	Q	季度

6.14 JSON/JSONB 函数和操作符

6.14.1 JSON/JSONB 操作符

表 6-12 json 和 jsonb 通用操作符

操作符	左操作数类型	右操作数类型	返回类型	描述	示例
->	Array-json(b)	int	json(b)	获得array-json元素。下标不存在返回空。	SELECT '[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2; ?column? ---------- {"c":"baz"} (1 row)
->	object-json(b)	text	json(b)	通过键获得值。不存在则返回空。	SELECT '{"a":{"b":"foo"}}'::json->'a'; ?column? ---------- {"b":"foo"} (1 row)
->>	Array-json(b)	int	text	获得array-json元素。下标不存在返回空。	SELECT '[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->>2; ?column? ---------- {"c":"baz"} (1 row)
->>	object-json(b)	text	text	通过键获得值。不存在则返回空	SELECT '{"a":{"b":"foo"}}'::json->>'a'; ?column? ---------- {"b":"foo"} (1 row)
#>	container-json(b)	text[]	json	获取在指定路径的 JSON 对象，路径不存在则返回空。 **说明** GaussDB(DWS)对象标识符支持以符号"#"结尾，为避免a#>b解析过程出现歧义，因此操作符"#>"前后需要增加空格，否则解析报错。	SELECT '{"a":{"b":{"c":1}}}'::json #> '{a, b}'; ?column? ---------- {"c":1} (1 row)
#>>	container-json(b)	text[]	text	获取在指定路径的 JSON 对象，路径不存在则返回空	SELECT '{"a":{"b":{"c":1}}}'::json #>> '{a, b}'; ?column? ---------- {"c":1} (1 row)

表 6-13 jsonb 支持的操作符

操作符	右操作类型	返回类型	描述	示例
=	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_eq	SELECT '{"a":{"b":{"c":1}}}'::jsonb = '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- t (1 row)
<>	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_ne	SELECT '{"a":{"b":{"c":1}}}'::jsonb <> '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)
<	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_lt	SELECT '{"a":{"b":{"c":2}}}'::jsonb < '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)
>	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_gt。	SELECT '{"a":{"b":{"c":2}}}'::jsonb > '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- t (1 row)
<=	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_le。	SELECT '{"a":{"b":{"c":2}}}'::jsonb <= '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- f (1 row)
>=	jsonb	bool	判断两个jsonb的大小关系。同函数jsonb_ge。	SELECT '{"a":{"b":{"c":2}}}'::jsonb >= '{"a":{"b":{"c":1}}}'::jsonb; ?column? ---------- t (1 row)
?	text	bool	键/元素的字符串是否存在JSON值的顶层	SELECT '{"a":1, "b":2}'::jsonb ? 'b'; ?column? ---------- t (1 row)
?|	text[]	bool	这些数组字符串中的任何一个是否作为顶层键存在。	SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb ?| '{a, b, e}'::text[]; ?column? ---------- t (1 row)
?&	text[]	bool	是否所有这些数组字符串都作为顶层键存在。	SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb ?& '{a, b, c}'::text[]; ?column? ---------- t (1 row)

操作符	右操作类型	返回类型	描述	示例
<@	jsonb	bool	左边的JSON的所有项是否全部存在于右边JSON的顶层。	SELECT '{"b":3}'::jsonb <@ '{"a":{"b":{"c":2}}, "b":3}'::jsonb; ?column? ---------- t (1 row)
@>	jsonb	bool	左边的JSON的顶层是否包含右边JSON的顶层所有项。	SELECT '{"a":{"b":{"c":2}}, "b":3}'::jsonb @> '{"b":3}'::jsonb; ?column? ---------- t (1 row)
||	jsonb	jsonb	两个jsonb对象合并成一个。	SELECT '{"a":1, "b":2}'::jsonb || '{"c":3, "d":4}'::jsonb; ?column? ---------- {"a": 1, "b": 2, "c": 3, "d": 4} (1 row)
-	text	jsonb	删除jsonb对象删除指定的键值对。	SELECT '{"a":1, "b":2}'::jsonb - 'a'; ?column? ---------- {"b": 2} (1 row)
-	text	jsonb	删除jsonb对象删除指定的键值对。	SELECT '{"a":1, "b":2, "c":3, "d":4}'::jsonb - '{a, b}'::text[]; ?column? ---------- {"c": 3, "d": 4} (1 row)
-	int	jsonb	删除jsonb数组中下标对应的元素。	SELECT '["a", "b", "c"]'::jsonb - 2; ?column? ---------- ["a", "b"] (1 row)
#-	text[]	jsonb	删除jsonb对象中路径对应的键值对。	SELECT '{"a":{"b":{"c":{"d":1}}}, "e":2, "f":3}'::jsonb #- '{a, b}'::text[]; ?column? ---------- {"a": {}, "e": 2, "f": 3} (1 row)

6.14.2 JSON/JSONB 函数

JSON/JSONB函数表示可以用于JSON类型（请参考**JSON类型**）数据的函数。

说明

除下列前两个函数array_to_json和row_to_json外，其余有关JSON/JSONB函数和操作符仅8.1.2及以上集群版本支持。

array_to_json(anyarray [, pretty_bool])

描述：返回JSON类型的数组。一个多维数组成为一个JSON数组的数组。如果pretty_bool为设置为true，将在一维元素之间添加换行符。

返回类型：json

示例：

```
SELECT array_to_json('{{1,5},{99,100}}'::int[]);
array_to_json
------------------
[[1,5],[99,100]]
(1 row)
```

row_to_json(record [, pretty_bool])

描述：返回JSON类型的行。如果pretty_bool设置为true，将在第一级元素之间添加换行符。

返回类型：json

示例：

```
SELECT row_to_json(row(1,'foo'));
    row_to_json
----------------------
 {"f1":1,"f2":"foo"}
(1 row)
```

json_agg(any)

描述：将值聚集为json数组。

返回类型：array-json

示例：

```
SELECT * FROM classes;
 name | score
 -----+-------
 A    |     2
 A    |     3
 D    |     5
 D    |
(4 rows)
SELECT name, json_agg(score) score FROM classes group by name order by name;
 name |      score
 -----+------------------
 A    | [2, 3]
 D    | [5, null]
      | [null]
(3 rows)
```

json_object_agg(any, any)

描述：将值聚集为json对象。

返回类型：json

示例：

```
SELECT * FROM classes;
 name | score
```



```
-----+-------
A   |    2
A   |    3
D   |    5
D   |
(4 rows)

SELECT json_object_agg(name, score) FROM classes group by name order by name;
    json_object_agg
--------------------------
{ "A" : 2, "A" : 3 }
{ "D" : 5, "D" : null }
(2 rows)
```

json_build_array(VARIADIC "any")

描述：从可变参数列表中构建一个可能是异构类型的JSON数组。

返回类型：json

示例：

```
SELECT json_build_array(1,2,'3',4,5);
 json_build_array
-------------------
 [1, 2, "3", 4, 5]
(1 row)
```

json_build_object(VARIADIC "any")

描述：从可变参数列表中构建JSON对象。参数列表由交替的键和值组成。其入参必须为偶数个，两两一组组成键值对。注意键不可为null。

返回类型：json

示例：

```
SELECT json_build_object('foo',1,'bar',2);
  json_build_object
------------------------
 {"foo" : 1, "bar" : 2}
(1 row)
```

json_object(text[])、json_object(text[], text[])

描述：从文本数组中构建JSON对象。

这是个重载函数，当入参为一个文本数组的时候，其数组长度必须为偶数，成员被当做交替出现的键/值对。两个文本数组的时候，第一个数组被视为键，第二个被视为值，两个数组长度必须相等。键不可为null。

返回类型：json

示例：

```
SELECT json_object('{a, 1, b, "def", c, 3.5}');
              json_object
---------------------------------------
 {"a" : "1", "b" : "def", "c" : "3.5"}
(1 row)

SELECT json_object('{{a, 1},{b, "def"},{c, 3.5}}');
              json_object
---------------------------------------
 {"a" : "1", "b" : "def", "c" : "3.5"}
```

```
(1 row)

SELECT json_object('{a,b,"a b c"}', '{a,1,1}');
                json_object
 ----------------------------------------
 {"a" : "a", "b" : "1", "a b c" : "1"}
(1 row)
```

to_json(anyelement)

描述：把参数转换为json。

返回类型：json

示例：

```
SELECT to_json('Fred said "Hi."'::text);
       to_json
---------------------
 "Fred said \"Hi.\""
(1 row)
——将列存表json_tbl_2转换为json
postgres=# SELECT * FROM json_tbl_2;
  a |  b
 ---+-----
  1 | aaa
  1 | bbb
  2 | ccc
  2 | ddd
 (4 rows)
postgres=# SELECT to_json(t.*) FROM json_tbl_2 t;
      to_json
-------------------
 {"a":1,"b":"bbb"}
 {"a":2,"b":"ddd"}
 {"a":1,"b":"aaa"}
 {"a":2,"b":"ccc"}
(4 rows)
```

json_strip_nulls(json)

描述：所有具有空值的对象字段被忽略，其他值保持不变。

返回类型：json

示例：

```
SELECT json_strip_nulls('[{"f1":1,"f2":null},2,null,3]');
  json_strip_nulls
---------------------
 [{"f1":1},2,null,3]
(1 row)
```

json_object_field(json, text)

描述：同操作符->, 返回对象中指定键对应的值。

返回类型：json

示例：

```
SELECT json_object_field('{"a": {"b":"foo"}}','a');
 json_object_field
-------------------
 {"b":"foo"}
(1 row)
```


json_object_field_text(object-json, text)

描述：同操作符->>, 返回对象中指定键对应的值。

返回类型：text

示例：

```
SELECT json_object_field_text('{"a": {"b":"foo"}}','a');
 json_object_field_text
------------------------
 {"b":"foo"}
(1 row)
```

json_array_element(array-json, integer)

描述：同操作符->, 返回数组中指定下标的元素。

返回类型：json

示例：

```
SELECT json_array_element('[1,true,[1,[2,3]],null]',2);
 json_array_element
--------------------
 [1,[2,3]]
(1 row)
```

json_array_element_text(array-json, integer)

描述：同操作符->>, 返回数组中指定下标的元素。

返回类型：text

示例：

```
SELECT json_array_element_text('[1,true,[1,[2,3]],null]',2);
 json_array_element_text
-------------------------
 [1,[2,3]]
(1 row)
```

json_extract_path(json, VARIADIC text[])

描述：同操作符#>，返回$2所指路径的JSON值。

返回类型：json

示例：

```
SELECT json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}', 'f4','f6');
 json_extract_path
-------------------
 "stringy"
(1 row)
```

json_extract_path_text(json, VARIADIC text[])

描述：同操作符#>>，返回$2所指路径的text值。

返回类型：text

示例：


```
SELECT json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}', 'f4','f6');
 json_extract_path_text
------------------------
 stringy
(1 row)
```

json_array_elements(array-json)

描述：拆分数组，每一个元素返回一行。

返回类型：json

示例：

```
SELECT json_array_elements('[1,true,[1,[2,3]],null]');
 json_array_elements
---------------------
 1
 true
 [1,[2,3]]
 null
(4 rows)
```

json_array_elements_text(array-json)

描述：拆分数组，每一个元素返回一行。

返回类型：text

示例：

```
SELECT * FROM json_array_elements_text('[1,true,[1,[2,3]],null]');
   value
-----------
 1
 true
 [1,[2,3]]

(4 rows)
```

json_array_length(array-json)

描述：返回数组长度。

返回类型：integer

示例：

```
SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4,null]');
 json_array_length
-------------------
         6
(1 row)
```

json_object_keys(object-json)

描述：返回对象中顶层的所有键。

返回类型：text

示例：

```
SELECT json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}, "f1":"abcd"}');
 json_object_keys
```



```
------------------
 f1
 f2
 f1
(3 rows)
```

json_each(object-json)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value json)

示例：

```
SELECT * FROM json_each('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |  value
-----+----------
 f1  | [1,2,3]
 f2  | {"f3":1}
 f4  | null
(3 rows)
```

json_each_text(object-json)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value text)

示例：

```
SELECT * FROM json_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |  value
-----+----------
 f1  | [1,2,3]
 f2  | {"f3":1}
 f4  |
(3 rows)
```

json_populate_record(anyelement, object-json [, bool])

描述：$1必须是一个复合类型的参数。将会把object-json里的每个对键值进行拆分，以键当做列名，与$1中的列名进行匹配查找，并填充到$1的格式中。

返回类型：anyelement

示例：

```
CREATE TYPE jpop  AS (a text, b INT, c timestamp);
SELECT * FROM json_populate_record(null::jpop,'{"a":"blurfl","x":43.2}');
   a    | b | c
--------+---+---
 blurfl |   |
(1 row)
```

json_populate_recordset(anyelement, array-json [, bool])

描述：参考函数json_populate_record、jsonb_populate_record，对$2数组的每一个元素进行上述参数函数的操作，因此这也要求$2数组的每个元素都是object-json类型。

返回类型：setof anyelement

示例：


```
CREATE TYPE jpop AS (a text, b INT, c timestamp);
SELECT * FROM json_populate_recordset(null::jpop, '[{"a":1,"b":2},{"a":3,"b":4}]');
 a | b | c
---+---+---
 1 | 2 |
 3 | 4 |
(2 rows)
```

json_to_record(object-json)

描述：正如所有返回record 的函数一样，调用者必须用一个AS子句显式地定义记录的结构。会将object-json的键值对进行拆分重组，把键当做列名，去匹配填充AS显示指定的记录的结构。

返回类型：record

示例：

```
SELECT * FROM json_to_record('{"a":1,"b":"foo","c":"bar"}'::json) as x(a int, b text, d text);
 a |  b  | d
---+-----+---
 1 | foo |
(1 row)
```

json_to_recordset(array-json)

描述：参考函数json_to_record，对数组内个每个元素，执行上述函数的操作，因此这要求数组内的每个元素都得是object-json。

返回类型：setof record

示例：

```
SELECT * FROM json_to_recordset('[{"a":1,"b":{"d":"foo"},"c":true},{"a":2,"c":false,"b":{"d":"bar"}}]') AS x(a
INT, b json, c BOOLEAN);
 a |      b      | c
---+-------------+---
 1 | {"d":"foo"} | t
 2 | {"d":"bar"} | f
(2 rows)

SELECT * FROM json_to_recordset('[{"a":1,"b":"foo","d":false},{"a":2,"b":"bar","c":true}]') AS x(a INT, b text, c
BOOLEAN);
 a |  b  | c
---+-----+---
 1 | foo |
 2 | bar | t
(2 rows)
```

json_typeof(json)

描述：检测json类型。

返回类型：text

示例：

```
SELECT value, json_typeof(value) from (values (json '123.4'), (json '"foo"'), (json 'true'), (json 'null'), (json
'[1, 2, 3]'), (json '{"x":"foo", "y":123}'), (NULL::json)) as data(value);
       value        | json_typeof
--------------------+-------------
 123.4              | number
 "foo"              | string
 true               | boolean
 null               | null
```

```
 [1, 2, 3]              | array
{"x":"foo", "y":123} | object
                       |
(7 rows)
```

jsonb_object(text[])

描述：从一个文本数组构造一个object-jsonb。这是个重载函数，当入参为一个文本数组的时候，其数组长度必须为偶数，成员被当做交替出现的键/值对。

返回类型：jsonb

示例：

```
SELECT jsonb_object('{a,1,b,2,3,NULL,"d e f","a b c"}');
                    jsonb_object
-----------------------------------------------------
 {"3": null, "a": "1", "b": "2", "d e f": "a b c"}
(1 row)
```

jsonb_object(text[], text[])

描述：两个文本数组的时候，第一个数组认为是键，第二个认为是值，两个数组长度必须相等。键不可为null。

返回类型：jsonb

示例：

```
SELECT jsonb_object('{a,b,"a b c"}', '{a,1,1}');
            jsonb_object
-------------------------------------
 {"a": "a", "b": "1", "a b c": "1"}
(1 row)
```

to_jsonb(anyment)

描述：将其他类型转换成对应的jsonb类型。

返回类型：jsonb

示例：

```
SELECT to_jsonb(1.1);
 to_jsonb
----------
 1.1
(1 row)
```

jsonb_agg

描述：将jsonb对象聚合成jsonb数组。

返回类型：jsonb

示例：

```
SELECT * FROM json_tbl_2;
 a |  b
---+-----
 1 | aaa
 1 | bbb
 2 | ccc
 2 | ddd
```



```
(4 rows)

SELECT a, jsonb_agg(b) FROM json_tbl_2 GROUP BY a ORDER BY a;
 a |   jsonb_agg
---+----------------
 1 | ["aaa", "bbb"]
 2 | ["ccc", "ddd"]
(2 rows)
```

jsonb_object_agg

描述：将键/值对聚集成一个JSON对象。

返回类型：jsonb

示例：

```
SELECT * FROM json_tbl_3;
 a |  b  | c
---+-----+----
 1 | aaa | 10
 1 | bbb | 20
 2 | ccc | 30
 2 | ddd | 40
(4 rows)
SELECT a, jsonb_object_agg(b, c) FROM json_tbl_3 GROUP BY a ORDER BY a;
 a |    jsonb_object_agg
---+------------------------
 1 | {"aaa": 10, "bbb": 20}
 2 | {"ccc": 30, "ddd": 40}
(2 rows)
```

jsonb_build_array([VARIADIC "any"])

描述：从一个可变参数列表构造一个可能包含异质类型的JSON数组。

返回类型：jsonb

示例：

```
SELECT jsonb_build_array('a',1,'b',1.2,'c',true,'d',null,'e',json '{"x": 3, "y": [1,2,3]}','');
                              jsonb_build_array
-------------------------------------------------------------------------------
 ["a", 1, "b", 1.2, "c", true, "d", null, "e", {"x": 3, "y": [1, 2, 3]}, null]
(1 row)
```

jsonb_build_object([VARIADIC “any”])

描述：从一个可变参数列表构造出一个JSON对象，其入参必须为偶数个，两两一组组成键值对。注意键不可为null。

返回类型：jsonb

示例：

```
SELECT jsonb_build_object(1,2);
 jsonb_build_object
--------------------
 {"1": 2}
(1 row)
```

jsonb_strip_nulls(jsonb)

描述：所有具有空值的对象字段均被省略。其他空值保持不变。

返回类型：jsonb

示例：

```
SELECT jsonb_strip_nulls('[{"f1":1,"f2":null},2,null,3]');
    jsonb_strip_nulls
--------------------------
 [{"f1": 1}, 2, null, 3]
(1 row)
```

jsonb_object_field(jsonb, text)

描述：同操作符->, 返回对象中指定键对应的值。

返回类型：jsonb

示例：

```
SELECT jsonb_object_field('{"a": {"b":"foo"}}','a');
 jsonb_object_field
--------------------
 {"b": "foo"}
(1 row)
```

jsonb_object_field_text(jsonb, text)

描述：同操作符->>, 返回对象中指定键对应的值。

返回类型：text

示例：

```
SELECT jsonb_object_field_text('{"a": {"b":"foo"}}','a');
 jsonb_object_field_text
-------------------------
 {"b": "foo"}
(1 row)
```

jsonb_array_element(array-jsonb, integer)

描述：同操作符->, 返回数组中指定下标的元素。

返回类型：jsonb

示例：

```
SELECT jsonb_array_element('[1,true,[1,[2,3]],null]',2);
 jsonb_array_element
---------------------
 [1, [2, 3]]
(1 row)
```

jsonb_array_element_text(array-jsonb, integer)

描述：同操作符->>, 返回数组中指定下标的元素。

返回类型：text

示例：

```
SELECT jsonb_array_element_text('[1,true,[1,[2,3]],null]',2);
 jsonb_array_element_text
--------------------------
```



```
 [1, [2, 3]]
(1 row)
```

jsonb_extract_path((jsonb, VARIADIC text[])

描述：等价于操作符#>，返回$2所指路径的值。

返回类型：jsonb

示例：

```
SELECT jsonb_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}', 'f4','f6');
 jsonb_extract_path
--------------------
 "stringy"
(1 row)
```

jsonb_extract_path_text((jsonb, VARIADIC text[])

描述：等价于操作符#>>，返回$2所指路径的值。

返回类型：text

示例：

```
SELECT jsonb_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}', 'f4','f6');
 jsonb_extract_path_text
-------------------------
 stringy
(1 row)
```

jsonb_array_elements(array-jsonb)

描述：拆分数组，每一个元素返回一行。

返回类型：jsonb

示例：

```
SELECT jsonb_array_elements('[1,true,[1,[2,3]],null]');
 jsonb_array_elements
----------------------
 1
 true
 [1, [2, 3]]
 null
(4 rows)
```

jsonb_array_elements_text(array-jsonb)

描述：拆分数组，每一个元素返回一行。

返回类型：text

示例：

```
SELECT * FROM jsonb_array_elements_text('[1,true,[1,[2,3]],null]');
    value
-------------
 1
 true
 [1, [2, 3]]

(4 rows)
```


jsonb_array_length(array-jsonb)

描述：返回数组长度。

返回类型：integer

示例：

```
SELECT jsonb_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4,null]');
 jsonb_array_length
--------------------
          6
(1 row)
```

jsonb_object_keys(object-jsonb)

描述：返回对象中顶层的所有键。

返回类型：SETOF text

示例：

```
SELECT jsonb_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}, "f1":"abcd"}');
 jsonb_object_keys
-------------------
 f1
 f2
(2 rows)
```

jsonb_each(object-jsonb)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value jsonb)

示例：

```
SELECT * FROM jsonb_each('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |   value
-----+-----------
 f1  | [1, 2, 3]
 f2  | {"f3": 1}
 f4  | null
(3 rows)
```

jsonb_each_text(object-jsonb)

描述：将对象的每个键值对拆分转换成一行两列。

返回类型：setof(key text, value text)

示例：

```
SELECT * FROM jsonb_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null}');
 key |   value
-----+-----------
 f1  | [1, 2, 3]
 f2  | {"f3": 1}
 f4  |
(3 rows)
```


jsonb_populate_record(anyelement, object-jsonb [, bool])

描述：$1必须是一个复合类型的参数。将会把object-json里的每个对键值进行拆分，以键当做列名，与$1中的列名进行匹配查找，并填充到$1的格式中。

返回类型：anyelement

示例：

```
SELECT * FROM jsonb_populate_record(null::jpop,'{"a":"blurfl","x":43.2}');
   a    | b | c
--------+---+---
 blurfl |   |
(1 row)
```

jsonb_populate_record_set(anyelement, array-jsonb [, bool])

描述：参考上述函数json_populate_record、jsonb_populate_record，对$2数组的每一个元素进行上述参数函数的操作，因此这也要求$2数组的每个元素都是object-json类型。

返回类型：setof anyelement

示例：

```
SELECT * FROM json_populate_recordset(null::jpop, '[{"a":1,"b":2},{"a":3,"b":4}]');
 a | b | c
---+---+---
 1 | 2 |
 3 | 4 |
(2 rows)
```

jsonb_to_record(object-json)

描述：正如所有返回record 的函数一样，调用者必须用一个AS子句显式地定义记录的结构。会将object-json的键值对进行拆分重组，把键当做列名，去匹配填充AS显示指定的记录的结构。

返回类型：record

示例：

```
SELECT * FROM jsonb_to_record('{"a":1,"b":"foo","c":"bar"}'::jsonb) as x(a int, b text, d text);
 a |  b  | d
---+-----+---
 1 | foo |
(1 row)
```

jsonb_to_recordset(array-json)

描述：参考函数jsonb_to_record，对数组内个每个元素，执行上述函数的操作，因此这要求数组内的每个元素都得是object-jsonb。

返回类型：setof record

示例：

```
SELECT * FROM jsonb_to_recordset('[{"a":1,"b":"foo","d":false},{"a":2,"b":"bar","c":true}]') AS x(a INT, b text,
c boolean);
 a |  b  | c
---+-----+---
 1 | foo |
```



```
 2 | bar | t
(2 rows)
```

jsonb_typeof(jsonb)

描述：检测jsonb类型。

返回类型：text

示例：

```
SELECT jsonb_typeof(to_jsonb(1.1));
 jsonb_typeof
--------------
 number
(1 row)
```

jsonb_ne(jsonb, jsonb)

描述：同操作符<>，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_ne('{"a":1, "b":2}'::jsonb, '{"a":1, "b":3}'::jsonb);
 jsonb_ne
----------
 t
(1 row)
```

jsonb_lt(jsonb, jsonb)

描述：同操作符<，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_lt('{"a":1, "b":2}'::jsonb, '{"a":1, "b":3}'::jsonb);
 jsonb_lt
----------
 t
(1 row)
```

jsonb_gt(jsonb, jsonb)

描述：同操作符>，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_gt('{"a":1, "b":2}'::jsonb, '{"a":1, "b":3}'::jsonb);
 jsonb_gt
----------
 f
(1 row)
```

jsonb_le(jsonb, jsonb)

描述：同操作符<=，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_le('["a", "b"]', '{"a":1, "b":2}');
 jsonb_le
----------
 t
(1 row)
```

jsonb_ge(jsonb, jsonb)

描述：同操作符>=，比较两个值的大小。

返回类型：bool

示例：

```
SELECT jsonb_ge('["a", "b"]', '{"a":1, "b":2}');
 jsonb_ge
----------
 f
(1 row)
```

jsonb_eq(jsonb, jsonb)

描述：同操作符=，比较两个值的大小

返回类型：bool

示例：

```
SELECT jsonb_eq('["a", "b"]', '{"a":1, "b":2}');
 jsonb_eq
----------
 f
(1 row)
```

jsonb_cmp(jsonb, jsonb)

描述：比较大小，正数表示大于，负数表示小于，0表示相等。

返回类型：integer

示例：

```
SELECT jsonb_cmp('["a", "b"]', '{"a":1, "b":2}');
 jsonb_cmp
-----------
 -1
(1 row)
```

jsonb_exists(jsonb, text)

描述：同操作符?，字符串$2是否存在$1的顶层以key\elem\scalar的形式存在。

返回类型：bool

示例：

```
SELECT jsonb_exists('["1",2,3]', '1');
 jsonb_exists
--------------
```



```
 t
(1 row)
```

jsonb_exists_any(jsonb, text[])

描述：同操作符?|，字符串数组$2里面是否存在的元素，在$1的顶层以key\elem\scalar的形式存在。

返回类型：

示例：

```
SELECT jsonb_exists_any('["1","2",3]', '{1, 2, 4}');
 jsonb_exists_any
------------------
 t
(1 row)
```

jsonb_exists_all(jsonb, text[])

描述：同操作符?&，字符串数组$2里面是否所有的元素，都在$1的顶层以key\elem\scalar的形式存在。

返回类型：

bool

示例：

```
SELECT jsonb_exists_all('["1","2",3]', '{1, 2}');
 jsonb_exists_all
------------------
 t
(1 row)
```

jsonb_contained(jsonb, jsonb)

描述：同操作符<@, 判断$1中的所有元素是否在$2的顶层存在。

返回类型：bool

示例：

```
SELECT jsonb_contained('[1,2,3]', '[1,2,3,4]');
 jsonb_contained
-----------------
 t
(1 row)
```

jsonb_contains(jsonb, jsonb)

描述：同操作符@>, 判断$1中的顶层所有元素是否包含在$2的所有元素。

返回类型：bool

示例：

```
SELECT jsonb_contains('{"a":1, "b":2, "c":3}'::jsonb, '{"a":1}');
 jsonb_contains
-----------------
 t
(1 row)
```


jsonb_concat(jsonb, jsonb)

描述：连接两个jsonb对象为一个jsonb。

返回类型：jsonb

示例：

```
SELECT jsonb_concat('{"a":1, "b":2}'::jsonb, '{"c":3, "d":4}'::jsonb);
          jsonb_concat
-----------------------------------
 {"a": 1, "b": 2, "c": 3, "d": 4}
(1 row)
```

jsonb_delete(jsonb, text)

描述：删除jsonb中的key值对应的键值对。

返回类型：jsonb

示例：

```
SELECT jsonb_delete('{"a":1, "b":2}'::jsonb, 'a');
 jsonb_delete
--------------
 {"b": 2}
(1 row)
```

jsonb_delete_idx(jsonb, text)

描述：删除数组下标对应的元素。

返回类型：jsonb

示例：

```
SELECT jsonb_delete_idx('[0,1,2,3,4]'::jsonb, 2);
 jsonb_delete_idx
------------------
 [0, 1, 3, 4]
(1 row)
```

jsonb_delete_array(jsonb, VARIADIC text[])

描述：删除jsonb数组中的多个元素。

返回类型：jsonb

示例：

```
SELECT jsonb_delete_array('["a", "b", "c"]'::jsonb , 'a', 'b');
 jsonb_delete_array
--------------------
 ["c"]
(1 row)
```

jsonb_delete_path(jsonb, text[])

描述：删除jsonb数组中指定路径的元素。

返回类型：jsonb

示例：


```
SELECT jsonb_delete_path('{"a":{"b":{"c":1, "d":2}}, "e":3}'::jsonb , array['a', 'b']);
 jsonb_delete_path
-------------------
 {"a": {}, "e": 3}
(1 row)
```

jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])

描述：返回target，用path指定的部分被new_value替换，或者如果create_missing为true(默认值为true)且path指定的项不存在，则添加new_value。与面向路径的运算符一样，path中出现的负整数从JSON数组的末尾开始计数。

返回类型：jsonb

示例：

```
SELECT jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false);
                jsonb_set
-----------------------------------------------
 [{"f1": [2, 3, 4], "f2": null}, 2, null, 3]
(1 row)
```

jsonb_pretty(jsonb)

描述：以缩进的JSON文本形式返回。

返回类型：jsonb

示例：

```
SELECT jsonb_pretty('{"a":{"b":{"c":1, "d":2}}, "e":3}'::jsonb);
    jsonb_pretty
---------------------
 {                  +
    "a": {          +
       "b": {       +
          "c": 1,+
          "d": 2 +
       }           +
    },             +
    "e": 3          +
 }
(1 row)
```

jsonb_insert(target jsonb, path text[], new_value jsonb [, insert_after boolean])

描述：返回target，并插入new_value。如果path指定的target部分位于JSONB数组中，则new_value将在目标之前或insert_after为 true(默认值为false)之后插入。如果在JSONB对象中由path指定的target部分，则仅当target不存在时才插入new_value。与面向路径的运算符一样，path中出现的负整数从JSON数组的末尾开始计数。

返回类型：jsonb

示例：

```
SELECT jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"');
         jsonb_insert
--------------------------------
 {"a": [0, "new_value", 1, 2]}
(1 row)
```


ts_headline([config regconfig,] document jsonb, query tsquery [, options text])

描述：高亮jsonb搜索结果。

返回类型：jsonb

示例：

```
SELECT ts_headline('english',
'[{"id":9928,"user_id":4562,"user_name":"9LOHR4","create_time":"2021-06-22T16:28:16.504518+08:00"},
{"id":9959,"user_id":5524,"user_name":"YID07D","create_time":"2021-06-22T16:28:16.557228+08:00"},
{"id":9962,"user_id":7991,"user_name":"7C6QOM","create_time":"2021-06-22T16:28:16.56234+08:00"}]'::jsonb,
 to_tsquery('english', '9LOHR4'), 'StartSel = <, StopSel = >');

ts_headline

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 [{"id": 9928, "user_id": 4562, "user_name": "<9LOHR4>", "create_time":
"2021-06-22T16:28:16.504518+08:00"}, {"id": 9959, "user_id": 5524, "user_name": "YID07D", "create_time":
"2021-06-22T16:28:16.557228+08:00"}, {"id": 9962, "user_id": 7991, "user_name": "7C6QOM",
"create_time": "2021-06-22T16:28:16.56234+08:00"}]
(1 row)
```

json_to_tsvector(config regconfig,] json, jsonb)

描述：将json格式转换为用于支持全文检索的文件格式tsvector。

返回类型：jsonb

示例：

```
SELECT json_to_tsvector('{"a":1, "b":2, "c":3}'::json, to_jsonb('key'::text));
 json_to_tsvector
------------------
 'b':2 'c':4
(1 row)
```

6.15 安全函数

gs_password_deadline()

描述：显示当前账户距离密码过期的时间。密码过期后提示用户修改密码。与GUC参数password_effect_time相关。

返回值类型：interval

示例：

```
SELECT gs_password_deadline();
  gs_password_deadline
-------------------------
 83 days 17:44:32.196094
(1 row)
```

gs_password_expiration()

描述：显示当前账户距离密码过期的时间。密码过期后用户无法登录数据库。与创建用户的DDL语句PASSWORD EXPIRATION period相关，函数返回值大于等于-1，如果

创建用户时未指定PASSWORD EXPIRATION period，该函数的缺省值为-1，表示没有过期限制。

返回值类型：interval

示例：

```
SELECT gs_password_expiration();
 gs_password_expiration
-------------------------
 29 days 23:59:49.731482
(1 row)
```

login_audit_messages(flag boolean)

描述：查看登录用户的登录信息。

返回值类型：元组

示例：

- 查看上一次登录认证通过的日期、时间和IP等信息：

```
SELECT * FROM login_audit_messages(true);
 username  | database |       logintime        |     type      | result |  client_conninfo
-----------+----------+------------------------+---------------+--------+--------------------
    dbadmin    | gaussdb | 2017-06-02 15:28:34+08 | login_success | ok     | gsql@[local]
(1 row)
```

- 查看上一次登录认证失败的日期、时间和IP等信息：

```
SELECT * FROM login_audit_messages(false) ORDER BY logintime desc limit 1;
 username  | database |       logintime        |     type      | result |     client_conninfo
-----------+----------+------------------------+---------------+--------+-------------------------
(0 rows)
```

- 查看自从最后一次认证通过以来失败的尝试次数、日期和时间：

```
SELECT * FROM login_audit_messages(false);
 username  | database |       logintime        |     type      | result |     client_conninfo
-----------+----------+------------------------+---------------+--------+-------------------------
(0 rows)
```

login_audit_messages_pid(flag boolean)

描述：查看登录用户的登录信息。与login_audit_messages的区别在于结果基于当前backendid向前查找。所以不会因为同一用户的后续登录，而影响本次登录的查询结果。也就是查询不到该用户后续登录的信息。

返回值类型：元组

示例：

- 查看上一次登录认证通过的日期、时间和IP等信息：

```
SELECT * FROM login_audit_messages_pid(true);
 username  | database |       logintime        |     type      | result |  client_conninfo | backendid
-----------+----------+------------------------+---------------+--------+--------------------
    dbadmin    | postgres | 2017-06-02 15:28:34+08 | login_success | ok     | gsql@[local] |
140311900702464
(1 row)
```

- 查看上一次登录认证失败的日期、时间和IP等信息：

```
SELECT * FROM login_audit_messages_pid(false) ORDER BY logintime desc limit 1;
 username  | database |       logintime        |     type      | result |     client_conninfo   | backendid
-----------+----------+------------------------+---------------+--------+-------------------------
(0 rows)
```

- 查看自从最后一次认证通过以来失败的尝试次数、日期和时间：


```
SELECT * FROM login_audit_messages_pid(false);
 username | database |       logintime        |     type     | result |     client_conninfo     | backendid
----------+----------+------------------------+--------------+--------+-------------------------+-----------
(0 rows)
```

pg_query_audit()

描述：查看当前CN节点审计日志。

返回值类型：record

函数返回字段如下表6-14：

表 6-14 pg_query_audit 函数返回字段

名称	类型	描述
begintime	timestamp with time zone	操作的执行开始时间。
endtime	timestamp with time zone	操作的执行结束时间。
operation_type	text	操作类型，具体类型见表6-15。
audit_type	text	审计类型，具体类型见表6-16。
result	text	操作结果。
username	text	执行操作的用户名。
database	text	数据库名称。
client_conninfo	text	客户端连接信息，即gsql，jdbc或odbc。
object_name	text	操作对象名称。
command_text	text	操作的执行命令。
detail_info	text	执行操作详细信息。
transaction_xid	text	事务ID。
query_id	text	查询ID。
node_name	text	节点名称。
thread_id	text	线程ID。
local_port	text	本地端口。
remote_port	text	远端端口。

表 6-15 operation_type 操作类型项

操作类型	描述
audit_switch	表示对用户打开和关闭审计日志操作场景进行审计。
login_logout	表示对用户登录和登出操作场景进行审计。
system	表示对系统的启停、实例切换操作场景进行审计。
sql_parse	表示对SQL语句解析场景进行审计。
user_lock	表示对用户锁定和解锁操作的场景进行审计。
grant_revoke	表示对用户权限授予和回收操作场景进行审计。
violation	表示对用户访问存在越权的场景进行审计。
ddl	表示对DDL操作场景进行审计，因为DDL操作会根据操作对象进行更细粒度控制，仍然沿用审计开关audit_system_object，即由audit_system_object控制对哪些对象的DDL操作进行审计（此处不配置ddl，只要配置了audit_system_object，审计也会生效)。
dml	表示对DML操作场景进行审计。
select	表示对SELECT操作场景进行审计。
internal_event	表示对内部事件操作场景进行审计。
user_func	表示对用户自定义函数、存储过程、匿名块操作场景进行审计。
special_func	表示对特殊函数调用操作场景进行审计，特殊函数包括：pg_terminate_backend和pg_cancel_backend。
copy	表示对COPY操作场景进行审计。
set	表示对SET操作场景进行审计。
transaction	表示对事务操作场景进行审计。
vacuum	表示对VACUUM操作场景进行审计。
analyze	表示对ANALYZE操作场景进行审计。
cursor	表示对游标操作的场景进行审计。
anonymous_block	表示对匿名块操作场景进行审计。
explain	表示对EXPLAIN操作场景进行审计。
show	表示对SHOW操作场景进行审计。
lock_table	表示对锁表操作场景进行审计。
comment	表示对COMMENT操作场景进行审计。
preparestmt	表示对PREPARE、EXECUTE、DEALLOCATE操作场景进行审计。

操作类型	描述
cluster	表示对CLUSTER操作场景进行审计。
constraints	表示对CONSTRAINTS操作场景进行审计。
checkpoint	表示对CHECKPOINT操作场景进行审计。
barrier	表示对BARRIER操作场景进行审计。
cleanconn	表示对CLEAN CONNECTION操作场景进行审计。
seclabel	表示对安全标签操作进行审计。
notify	表示对通知操作进行审计。
load	表示对加载操作进行审计。

表 6-16 audit_type 审计类型项

审计类型	描述
audit_open/audit_close	表示审计类型为打开和关闭审计日志操作。
user_login/user_logout	表示审计类型为用户登录/退出成功的操作和用户。
system_start/system_stop/ system_recover/system_switch	表示审计类型为系统的启停、实例切换操作。
sql_wait/sql_parse	表示审计类型为SQL语句解析。
lock_user/unlock_user	表示审计类型为用户锁定和解锁成功的操作。
grant_role/revoke__role	表示审计类型为用户权限授予和回收的操作。
user_violation	表示审计类型为用户访问存在越权的操作。
ddl_数据库对象	表示审计类型为DDL操作，因为DDL操作由会根据操作对象进行更细粒度控制，仍然沿用审计开关audit_system_object，即由audit_system_object控制对哪些对象的DDL操作进行审计（此处不配置ddl，只要配置了audit_system_object，审计也会生效）。 例如：ddl_sequence表示审计类型为序列相关操作。
dml_action_insert/ dml_action_delete/ dml_action_update/ dml_action_merge/ dml_action_select	表示审计类型为INSERT、DELETE、UPDATE、MERGE等DML操作。
internal_event	表示审计类型为内部事件。

审计类型	描述
user_func	表示审计类型为用户自定义函数、存储过程、匿名块操作。
special_func	表示审计类型为特殊函数调用操作，特殊函数包括：pg_terminate_backend和pg_cancel_backend。
copy_to/copy_from	表示审计类型为COPY相关操作。
set_parameter	表示审计类型为SET操作。
trans_begin/trans_commit/ trans_prepare/ trans_rollback_to/ trans_release/trans_savepoint/ trans_commit_prepare/ trans_rollback_prepare/ trans_rollback	表示审计类型为事务相关操作。
vacuum/vacuum_full/ vacuum_merge	表示审计类型为VACUUM相关操作。
analyze/analyze_verify	表示审计类型为ANALYZE相关操作。
cursor_declare/cursor_move/ cursor_fetch/cursor_close	表示审计类型为游标相关操作。
codeblock_execute	表示审计类型为匿名块。
explain	表示审计类型为EXPLAIN操作。
show	表示审计类型为SHOW操作。
lock_table	表示审计类型为锁表操作。
comment	表示审计类型为COMMENT操作。
prepare/execute/deallocate	表示审计类型为PREPARE、EXECUTE或DEALLOCATE操作。
cluster	表示审计类型为CLUSTER操作。
constraints	表示审计类型为CONSTRAINTS操作。
checkpoint	表示审计类型为CHECKPOINT操作。
barrier	表示审计类型为BARRIER操作。
cleanconn	表示审计类型为CLEAN CONNECTION操作。
seclabel	表示审计类型为安全标签操作。
notify	表示审计类型为通知操作。
load	表示审计类型为加载操作。

pgxc_query_audit()

描述：查看所有CN节点审计日志。

返回值类型：record

函数返回字段同**pg_query_audit()**函数。

pg_delete_audit()

描述：删除指定时间段的审计日志。 返回值类型：void

说明

基于数据库安全考虑，不提供删除指定时间段的审计日志的函数接口，调用该函数将直接报“ERROR: For security purposes, it is not allowed to manually delete audit logs”。

```
SELECT * FROM pg_delete_audit('2023-01-10 17:00:00','2023-01-10 19:00:00');
ERROR:  For security purposes, it is not allowed to manually delete audit logs
```

6.16 条件表达式函数

coalesce(expr1, expr2, ..., exprn)

描述：返回参数列表中第一个非NULL的参数值。

COALESCE(expr1, expr2) 等价于CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END。

示例：

```
SELECT coalesce(NULL,'hello');
 coalesce
----------
 hello
(1 row)
```

说明

- 如果表达式列表中的所有表达式都等于NULL，则本函数返回NULL。
- 它常用于在显示数据时用缺省值替换NULL。
- 和CASE表达式一样，COALESCE不会计算不需要用来判断结果的参数；即在第一个非空参数右边的参数不会被计算。

decode(base_expr, compare1, value1, Compare2,value2, … default)

描述：把base_expr与后面的每个compare(n) 进行比较，如果匹配返回相应的value(n)。如果没有发生匹配，则返回default。

示例：

```
SELECT decode('A','A',1,'B',2,0);
 case
------
 1
(1 row)
```

if(bool_expr, expr1, expr2)

描述：当bool_expr为true时，返回expr1，否则返回expr2。

if(bool_expr, expr1, expr2) 等价于CASE WHEN bool_expr = true THEN expr1 ELSE expr2 END。

示例：

```
SELECT if(1 < 2, 'yes', 'no');
 if
-----
 yes
(1 row)
```

说明

参数expr1和expr2可以为任意类型，返回结果类型规则请参考**UNION，CASE和相关构造**。

ifnull(expr1, expr2)

描述：当expr1不为NULL时，返回expr1，否则返回expr2。

ifnull(expr1, expr2) 逻辑上等价于CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END。

示例：

```
SELECT ifnull(NULL,'hello');
 ifnull
--------
 hello
(1 row)
```

说明

参数expr1和expr2可以为任意类型，返回结果类型规则请参考**UNION，CASE和相关构造**。

isnull(expr)

描述：当expr为NULL时，返回true，否则返回false。

isnull(expr) 逻辑上等价于expr IS NULL。

示例：

```
SELECT isnull(NULL), isnull('abc');
 isnull | isnull
--------+--------
 t      | f
(1 row)
```

nullif(expr1, expr2)

描述：当且仅当expr1和expr2相等时，NULLIF才返回NULL，否则它返回expr1。

nullif(expr1, expr2) 逻辑上等价于CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END。

示例：

```
SELECT nullif('hello','world');
 nullif
--------
 hello
(1 row)
```

备注：

如果两个参数的数据类型不同，则：

- 两种数据类型之间存在隐式转换，则以其中优先级较高的数据类型为基准将另一个参数隐式转换成该类型，转换成功则进行计算，转换失败则返回错误。如：

```
SELECT nullif('1234'::VARCHAR,123::INT4);
 nullif
--------
   1234
(1 row)
SELECT nullif('1234'::VARCHAR,'2012-12-24'::DATE);
ERROR:  invalid input syntax for type timestamp: "1234"
```

- 两种数据类型之间不存在隐式转换，则返回错误 。如：

```
SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE);
ERROR:  operator does not exist: boolean = timestamp without time zone
LINE 1: SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE) FROM DUAL;
^
HINT:  No operator matches the given name and argument type(s). You might need to add explicit type casts.
```

nvl(expr1 , expr2)

描述：如果expr1为NULL则返回expr2。如果expr1非NULL，则返回expr1。

示例：

```
SELECT nvl('hello','world');
  nvl
-------
 hello
(1 row)
```

说明

参数expr1和expr2可以为任意类型，当NVL的两个参数不属于同类型时，看第二个参数是否可以向第一个参数进行隐式转换，如果可以则返回第一个参数类型。如果第二个参数不能向第一个参数进行隐式转换而第一个参数可以向第二个参数进行隐式转换，则返回第二个参数的类型。如果两个参数之间不存在隐式类型转换并且也不属于同一类型则报错。

sys_context('namespace' , 'parameter')

描述：获取并返回指定namespace下参数parameter的值。

返回值类型：VARCHAR

示例：

```
SELECT sys_context('USERENV', 'CURRENT_SCHEMA');
 sys_context
-------------
 public
(1 row)
```

根据当前所在的实际schema而变化。

说明

目前仅支持SYS_CONTEXT('USERENV', 'CURRENT_SCHEMA') 和SYS_CONTEXT('USERENV', 'CURRENT_USER')两种格式。

greatest(expr1 [, ...])

描述：获取并返回参数列表中值最大的表达式的值。

- ORA和TD兼容模式下，返回结果为所有非null参数的最大值。
- MySQL兼容模式下，入参中存在null时，返回结果为null。

示例：

```
SELECT greatest(1*2,2-3,4-1);
 greatest
----------
        3
(1 row)
SELECT greatest('ABC', 'BCD', 'CDE');
 greatest
----------
 CDE
(1 row)
```

least(expr1 [, ...])

描述：获取并返回参数列表中值最小的表达式的值。

- ORA和TD兼容模式下，返回结果为所有非null参数的最小值。
- MySQL兼容模式下，入参中存在null时，返回结果为null。

示例：

```
SELECT least(1*2,2-3,4-1);
 least
-------
    -1
(1 row)
SELECT least('ABC','BCD','CDE');
 least
--------
 ABC
(1 row)
```

EMPTY_BLOB()

描述：使用EMPTY_BLOB在INSERT或UPDATE语句中初始化一个BLOB变量，取值为NULL。

返回值类型：BLOB

示例：

```
--新建表
CREATE TABLE blob_tb(b blob,id int) DISTRIBUTE BY REPLICATION;
--插入数据
INSERT INTO blob_tb VALUES (empty_blob(),1);
--删除表
 DROP TABLE blob_tb;
```

说明

使用DBMS.GETLENGTH求得的长度为0。

6.17 范围函数和操作符

6.17.1 范围操作符

=

描述：等于

示例：

```
SELECT int4range(1,5) = '[1,4]'::int4range AS RESULT;
 result
--------
 t
(1 row)
```

<>

描述：不等于

示例：

```
SELECT numrange(1.1,2.2) <> numrange(1.1,2.3) AS RESULT;
 result
--------
 t
(1 row)
```

<

描述：小于

示例：

```
SELECT int4range(1,10) < int4range(2,3) AS RESULT;
 result
--------
 t
(1 row)
```

>

描述：大于

示例：

```
SELECT int4range(1,10) > int4range(1,5) AS RESULT;
 result
--------
 t
(1 row)
```

<=

描述：小于或等于

示例：

```
SELECT numrange(1.1,2.2) <= numrange(1.1,2.2) AS RESULT;
 result
--------
 t
(1 row)
```


>=

描述：大于或等于

示例：

```
SELECT numrange(1.1,2.2) >= numrange(1.1,2.0) AS RESULT;
 result
--------
 t
(1 row)
```

@>

描述：包含范围、包含元素

示例：

```
SELECT int4range(2,4) @> int4range(2,3) AS RESULT;
 result
--------
 t
(1 row)
SELECT '[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp AS RESULT;
 result
--------
 t
(1 row)
```

<@

描述：范围包含于、元素包含于

示例：

```
SELECT int4range(2,4) <@ int4range(1,7) AS RESULT;
 result
--------
 t
(1 row)
SELECT 42 <@ int4range(1,7) AS RESULT;
 result
--------
 f
(1 row)
```

&&

描述：重叠（有共同点）

示例：

```
SELECT int8range(3,7) && int8range(4,12) AS RESULT;
 result
--------
 t
(1 row)
```

<<

描述：范围值是否比另一个范围值的最小值还小（没有交集）

示例：


```
SELECT int8range(1,10) << int8range(100,110) AS RESULT;
 result
--------
 t
(1 row)
```

>>

描述：范围值是否比另一个范围值的最大值还大（没有交集）

示例：

```
SELECT int8range(50,60) >> int8range(20,30) AS RESULT;
 result
--------
 t
(1 row)
```

&<

描述：范围值的最大值是否不超过另一个范围值的最大值。

示例：

```
SELECT int8range(1,20) &< int8range(18,20) AS RESULT;
 result
--------
 t
(1 row)
```

&>

描述：范围值的最小值是否不小于另一个范围值的最小值。

示例：

```
SELECT int8range(7,20) &> int8range(5,10) AS RESULT;
 result
--------
 t
(1 row)
```

-|-

描述：相邻

示例：

```
SELECT numrange(1.1,2.2) -|- numrange(2.2,3.3) AS RESULT;
 result
--------
 t
(1 row)
```

+

描述：并集

示例：

```
SELECT numrange(5,15) + numrange(10,20) AS RESULT;
 result
--------
```



```
 [5,20)
(1 row)
```

*****

描述：交集

示例：

```
SELECT int8range(5,15) * int8range(10,20) AS RESULT;
 result
---------
 [10,15)
(1 row)
```

-

描述：差集

示例：

```
SELECT int8range(5,15) - int8range(10,20) AS RESULT;
 result
--------
 [5,10)
(1 row
```

说明

- 简单的比较操作符<，>，<=和>=先比较下界，只有下界相等时才比较上界。
- <<、>>和-|-操作符当包含空范围时也会返回false；也就是，不认为空范围在其他范围之前或之后。
- 并集和差集操作符的执行结果无法包含两个不相交的子范围。

6.17.2 范围函数

lower(anyrange)

描述：范围的下界

返回类型：范围元素类型

示例：

```
SELECT lower(numrange(1.1,2.2)) AS RESULT;
 result
--------
    1.1
(1 row)
```

upper(anyrange)

描述：范围的上界

返回类型：范围元素类型

示例：

```
SELECT upper(numrange(1.1,2.2)) AS RESULT;
 result
--------
```



```
   2.2
(1 row)
```

isempty(anyrange)

描述：范围是否为空

返回类型：boolean

示例：

```
SELECT isempty(numrange(1.1,2.2)) AS RESULT;
 result
--------
 f
(1 row)
```

lower_inc(anyrange)

描述：是否包含下界

返回类型：boolean

示例：

```
SELECT lower_inc(numrange(1.1,2.2)) AS RESULT;
 result
--------
 t
(1 row)
```

upper_inc(anyrange)

描述：是否包含上界

返回类型：boolean

示例：

```
SELECT upper_inc(numrange(1.1,2.2)) AS RESULT;
 result
--------
 f
(1 row)
```

lower_inf(anyrange)

描述：下界是否为无穷

返回类型：boolean

示例：

```
SELECT lower_inf('(,)'::daterange) AS RESULT;
 result
--------
 t
(1 row)
```

upper_inf(anyrange)

描述：上界是否为无穷

返回类型：boolean

示例：

```
SELECT upper_inf('(,)'::daterange) AS RESULT;
 result
--------
 t
(1 row)
```

说明

如果范围是空或者需要的界限是无穷的，lower和upper函数将返回null。lower_inc、upper_inc、lower_inf和upper_inf函数均对空范围返回false。

6.18 数据脱敏函数

数据脱敏函数提供一系列不同脱敏形式的函数接口，可以覆盖常见的脱敏场景。通常结合数据脱敏语法，与脱敏列绑定使用，不推荐直接作用在查询语句上。

mask_none(column_name)

描述：不作任何脱敏处理，仅内部测试用。

返回值类型：与入参column_name数据类型相同。

mask_full(column_name)

描述：全脱敏成固定值。脱敏列的数据类型不同，脱敏的固定值不同。

返回值类型：与入参column_name数据类型相同。

mask_partial(column_name, mask_digital, mask_from[, mask_to])

描述：针对数值类型数据，将第mask_from到mask_to位的数字部分脱敏成mask_digital对应的数字。其中，参数mask_to允许缺省，缺省时即脱敏到数据结束位置。参数mask_digital只能取[0,9]区间内的数字。

返回值类型：与入参column_name数据类型相同。

mask_partial(column_name [, input_format, output_format], mask_char, mask_from[, mask_to])

描述：针对字符类型数据，对照指定的输入输出格式，将第mask_from到mask_to位的数字部分脱敏成mask_char指定的字符。

参数说明：

- input_format

  输入格式是由V和F组成的字符序列，与脱敏列数据长度相同。V对应位置的字符可能会被脱敏，F对象位置的字符会被忽略跳过，V字符序列标识脱敏范围。输入输出格式参数适用于定长数据，比如，银行卡号、身份证号、手机号等。

- output_format

  输出格式是由V和其他任意字符组成的字符序列，与脱敏列数据长度相同。V字符位置与input_format的V位置对应，其他字符位置与input_format的F位置对应，且不会脱敏，通常为数据分隔符。

input_format和output_format可以缺省或指定为空串""，此时，无输入输出格式要求，原始字符序列范围即为脱敏范围。

- mask_char
  脱敏字符，仅允许长度为1的任意字符。场景的脱敏字符包括"*"，"#"等。
- mask_from
  脱敏范围的起始位置，要求大于0。
- mask_to
  脱敏范围的结束位置，允许缺省。缺省时，即脱敏到原始数据结束位置。

返回值类型：与入参column_name数据类型相同。

mask_partial(column_name, mask_field1, mask_value1, mask_field2, mask_value2, mask_field3, mask_value3)

描述：按指定三个时间域做部分脱敏，仅适用于日期或时间类型数据。若mask_value取-1，即此mask_field不脱敏。其中，mask_field可以取"month"、"day"、"year"、"hour"、"minute"、"second"六个时间域之一。各域的取值范围需满足实际时间单位的取值范围。

返回值类型：与入参column_name数据类型相同。

说明

脱敏函数可以覆盖常见敏感信息的脱敏场景，推荐用户优先使用脱敏函数创建脱敏策略。
数据脱敏函数的使用方法，请参考**数据脱敏**中的示例。

自定义脱敏函数

支持用户使用PL/PGSQL语言自定义脱敏函数。

自定义脱敏函数需要严格遵循如下要求：

- 返回值与脱敏列类型一致。
- 函数必须可下推。
- 参数列表除脱敏格式外，只能包含一个脱敏列。
- 函数仅实现针对特定数据类型的格式化改写功能，不能涉及与其他表对象的复杂关联操作。

不满足前两项中任一项时，创建脱敏策略会报错。不满足后两项中任一项时，可成功创建脱敏策略，但查询执行结果可能会出现不可预知的问题。

6.19 Roaring Bitmap 函数和操作符

6.19.1 Roaring Bitmap 操作符

GaussDB(DWS)自8.1.3集群版本开始，支持高效的位图处理函数和操作符，可用于用户画像，精准营销等场景，极大的提高了查询性能。

=

描述：比较两个roaringbitmap是否相等。

返回值类型：bool

示例：

```
SELECT rb_build('{1,2,3}') = rb_build('{1,2,3}');
?column?
----------
t
(1 row)
SELECT rb_build('{2,3}') = rb_build('{1,2,3}');
?column?
----------
f
(1 row)
```

<>

描述：比较两个roaringbitmap是否不相等。

返回值类型：bool

示例：

```
SELECT rb_build('{1,2,3}') <> rb_build('{1,2,3}');
?column?
----------
f
(1 row)
SELECT rb_build('{2,3}') <> rb_build('{1,2,3}');
?column?
----------
t
(1 row)
```

&

描述：计算两个roaringbitmap交集以后的结果。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_build('{2,3}') & rb_build('{1,2,3}'));
rb_to_array
-------------
{2,3}
(1 row)
```

|

描述：计算两个roaringbitmap并集后的结果。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_build('{2,3}') | rb_build('{1,2,3}'));
rb_to_array
-------------
{1,2,3}
(1 row)
```

|

描述：计算一个roaringbitmap增加一个ID后的结果。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_build('{2,3}') | 4);
rb_to_array
-------------
{2,3,4}
(1 row)
```

#

描述：两个roaringbitmap进行异或后的结果。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_build('{2,3}') # rb_build('{1,2,3}'));
rb_to_array
-------------
{1}
(1 row)
```

-

描述：计算属于第一个roaringbitmap，但是不属于第二个roaringbitmap的结果集。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_build('{2,3,4}') - rb_build('{1,2,3}'));
rb_to_array
-------------
{4}
(1 row)
```

-

描述：从roaringbitmap中去掉一个指定ID后的结果集。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_build('{2,3,4}') - 3);
rb_to_array
-------------
{2,4}
(1 row)
```

@>

描述：操作符前面的roaringbitmap是否包含后面的roaringbitmap。

返回值类型：bool

示例：

```
SELECT rb_build('{2,3,4}') @> rb_build('{2,3}');
?column?
----------
t
```



```
(1 row)
SELECT rb_build('{2,3,4}') @> 4;
?column?
----------
t
(1 row)
```

<@

描述：操作符前面的roaringbitmap是否被后面的roaringbitmap包含。

返回值类型：bool

示例：

```
SELECT  4 <@ rb_build('{2,3,4}');
?column?
----------
t
(1 row)
SELECT rb_build('{2,3,4}') <@ rb_build('{2,3}');
?column?
----------
f
(1 row)
```

&&

描述：两个roaringbitmap如果有交集返回true，否则返回false。

返回值类型：bool

示例：

```
SELECT rb_build('{2,3,4}') && rb_build('{2,3}');
?column?
----------
t
(1 row)
SELECT rb_build('{2,3,4}') && rb_build('{7,8,9}');
?column?
----------
f
(1 row)
```

6.19.2 Roaring Bitmap 函数

GaussDB(DWS)自8.1.3集群版本开始，支持高效的位图处理函数和操作符，可用于用户画像，精准营销等场景，极大的提高了查询性能。

rb_build(array)

描述：将int数组转成roaringbitmap类型。

返回值类型：roaringbitmap

示例：

```
SELECT rb_build('{1,2,3}');
rb_build
------------------------------------------------
\x3a300000010000000000020010000000010002000300
(1 row)
CREATE TABLE r_row (a int, b text, c roaringbitmap);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using round-robin as the distribution mode by default.
```



```
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

INSERT INTO r_row values (1, 'a', rb_build('{1,2,3}'));
INSERT 0 1

SELECT * FROM r_row;
 a | b |                         c
---+---+-----------------------------------------------------
 1 | a | \x3a300000010000000000002001000000010002000300
(1 row)

INSERT INTO r_row values (2, 'b', rb_build('{}'));
INSERT 0 1

SELECT * FROM r_row;
 a | b |                         c
---+---+-----------------------------------------------------
 2 | b | \x3a30000000000000
 1 | a | \x3a300000010000000000002001000000010002000300
(2 rows)
```

rb_iterate(roaringbitmap)

描述：把roaringbitmap数据转成int，按照多行输出。

返回值类型：record类型(多行int值)

示例：

```
SELECT rb_iterate(c) FROM r_row;
rb_iterate
------------
1
2
3
(3 rows)
```

rb_to_array(roaringbitmap)

描述：rb_build的逆向操作，把roaringBitmap转成int数组。

返回值类型: array

示例：

```
SELECT rb_to_array(c) FROM r_row;
rb_to_array
-------------
{1,2,3}
(1 row)
SELECT rb_to_array('\x3a300000010000000000002001000000010002000300');
rb_to_array
-------------
{1,2,3}
(1 row)
```

rb_and(roaringbitmap, roaringbitmap)

描述：计算两个roaringbitmap的交集。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_and(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
```



```
-------------
{2,3}
(1 row)
```

rb_or(roaringbitmap, roaringbitmap)

描述：计算两个roaringbitmap的并集。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_or(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
-------------
{1,2,3,4}
(1 row)
```

rb_xor(roaringbitmap, roaringbitmap)

描述：计算两个roaringbitmap的异或。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_xor(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
-------------
{1,4}
(1 row)
```

rb_andnot(roaringbitmap, roaringbitmap)

描述：在第一个roaringbitmap集合中，但是不在第二个roaringbitmap中的集合。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_andnot(rb_build('{1,2,3}'), rb_build('{2,3,4}')));
rb_to_array
-------------
{1}
(1 row)
```

rb_cardinality(roaringbitmap)

描述：计算一个roaringbitmap的基数。

返回值类型：int

示例：

```
SELECT rb_cardinality(rb_build('{1,2,3}'));
rb_cardinality
----------------
3
(1 row)
```

rb_and_cardinality(roaringbitmap, roaringbitmap)

描述：计算两个roaringbitmap的交集的基数。

返回值类型：int

示例：

```
SELECT rb_and_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_and_cardinality
--------------------
2
(1 row)
```

rb_or_cardinality(roaringbitmap, roaringbitmap)

描述：计算两个roaringbitmap的并集的基数。

返回值类型：int

示例：

```
SELECT rb_or_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_or_cardinality
-------------------
4
(1 row)
```

rb_xor_cardinality(roaringbitmap, roaringbitmap)

描述：计算两个roaringbitmap异或以后的基数。

返回值类型：int

示例：

```
SELECT rb_xor_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_xor_cardinality
--------------------
2
(1 row)
```

rb_andnot_cardinality(roaringbitmap, roaringbitmap)

描述：计算两个roaringbitmap按照andnot计算结果以后的基数。

返回值类型：int

示例：

```
SELECT rb_andnot_cardinality(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_andnot_cardinality
-----------------------
1
(1 row)
```

rb_is_empty(roaringbitmap)

描述：判断一个roaringbitmap是否为空。

返回值类型：bool

示例：

```
SELECT rb_is_empty(rb_build('{1,2,3}'));
rb_is_empty
-------------
```



```
f
(1 row)
```

rb_equals(roaringbitmap, roaringbitmap)

描述：判断两个roaringbitmap是否相等。

返回值类型：bool

示例：

```
SELECT rb_equals(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_equals
-----------
f
(1 row)
```

rb_intersect(roaringbitmap, roaringbitmap)

描述：判断两个roaringbitmap是否相交。

返回值类型：bool

示例：

```
SELECT rb_intersect(rb_build('{1,2,3}'), rb_build('{2,3,4}'));
rb_intersect
--------------
t
(1 row)
```

rb_min(roaringbitmap)

描述：返回roaringbitmap中的最小值。

返回值类型：int

示例：

```
SELECT rb_min(rb_build('{1,2,3}'));
rb_min
--------
1
(1 row)
```

rb_max(roaringbitmap)

描述：返回roaringbitmap中的最大值。

返回值类型：int

示例：

```
SELECT rb_max(rb_build('{1,2,3}'));
rb_max
--------
3
(1 row)
```

rb_add(roaringbitmap, int)

描述：在roaringbitmap中增加一个元素。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_add(rb_build('{1,3}'), 2));
rb_to_array
-------------
{1,2,3}
(1 row)
```

rb_added(int, roaringbitmap)

描述：在roaringbitmap中增加一个元素。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_added(2, rb_build('{1,3}')));
rb_to_array
-------------
{1,2,3}
(1 row)
```

rb_contain(roaringbitmap,int)

描述：判断roaringbitmap是否包含指定的元素。

返回值类型：bool

示例：

```
SELECT rb_contain(rb_build('{1,3}'), 2);
rb_contain
------------
f
(1 row)
```

rb_containedby(int,roaringbitmap)

描述：判断给定的元素是否被给定的roaringbitmap包含。

示例：

```
SELECT rb_containedby(2,rb_build('{1,3}'));
rb_containedby
----------------
f
(1 row)
```

rb_contain_rb(roaringbitmap,roaringbitmap)

描述：判断第一个roaringbitmap是否包含第二个roaringbitmap。

返回值类型：bool

示例：

```
SELECT rb_contain_rb(rb_build('{1,3}'), rb_build('{2,3}'));
rb_contain_rb
---------------
f
(1 row)
```


rb_containedby_rb(roaringbitmap,roaringbitmap)

描述：判断跟定的第二个roaringbitmap是否包含第一个roaringbitmap。

返回值类型：bool

示例：

```
SELECT rb_containedby_rb(rb_build('{1,3}'), rb_build('{2,3}'));
rb_containedby_rb
----------------
f
(1 row)
```

rb_remove(roaringbitmap,int)

描述：从roaringbitmap中移除指定的元素。

返回值类型：roaringbitmap

示例：

```
SELECT rb_to_array(rb_remove(rb_build('{1,3}'),1));
rb_to_array
-------------
{3}
(1 row)
```

rb_clear(roaringbitmap,int,int)

描述：从roaringbitmap中清除指定范围内的元素。

返回值类型：roaringbitmap

示例：

```
SELECT
rb_to_array(rb_clear(rb_build('{1,2,3}'),1,2));

rb_to_array

-------------

{2,3}
                                                                (1 row)
```

rb_flip(roaringbitmap,int,int)

描述：反转指定范围的元素。

示例：

```
SELECT rb_to_array(rb_flip(rb_build('{1,2,3,7,9}'), 1,10));
rb_to_array
--------------
{4,5,6,8,10}
(1 row)
```

rb_rank(roaringbitmap,int)

描述：返回小于指定值的集合的基数。

返回值类型：int

示例：

```
SELECT rb_rank(rb_build('{1,10,100}'),99);
rb_rank
---------
2
(1 row)
```

6.19.3 Roaring Bitmap 聚合函数

GaussDB(DWS)自8.1.3集群版本开始，支持高效的位图聚合函数，可用于用户画像，精准营销等场景，极大的提高了查询性能。

rb_build_agg(int)

描述：将分组内的int值聚合成一个roaringbitmap值。

返回值类型：roaringbitmap

示例：

```
CREATE TABLE t1 (a int ,b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using round-robin as the distribution mode by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
INSERT INTO t1 SELECT generate_series(1,10),generate_series(1,20,2);
INSERT 0 10
SELECT rb_iterate(rb_build_agg(b)) FROM t1;
rb_iterate
------------
1
3
5
7
9
11
13
15
17
19
(10 rows)
```

rb_and_agg(roaringbitmap)

描述：将分组内的roaringbitmap数据按照交的操作聚合成一个roaringbitmap集合。

示例：

```
CREATE TABLE r1(a int ,b roaringbitmap);
INSERT INTO r1 SELECT a, rb_build_agg(b) FROM t1 GROUP BY a;
INSERT INTO t1 SELECT generate_series(1,10),generate_series(1,20,4);
INSERT INTO r1 SELECT a, rb_build_agg(b) FROM t1 GROUP BY a;
SELECT a, rb_to_array(rb_and_agg(b)) FROM r1 GROUP BY a ORDER BY a;
 a  | rb_to_array
----+-------------
 1  | {1}
 2  | {3}
 3  | {5}
 4  | {7}
 5  | {9}
 6  | {11}
 7  | {13}
 8  | {15}
 9  | {17}
10  | {19}
(10 rows)
```

rb_or_agg(roaringbitmap)

描述：将分组内的roaringbitmap按照并的逻辑组合成一个roaringbitmap。

示例：

```
SELECT a, rb_to_array(rb_or_agg(b)) FROM r1 GROUP BY a ORDER BY a;
 a  | rb_to_array
----+-------------
 1  | {1}
 2  | {3,5}
 3  | {5,9}
 4  | {7,13}
 5  | {9,17}
 6  | {1,11}
 7  | {5,13}
 8  | {9,15}
 9  | {13,17}
 10 | {17,19}
(10 rows)
```

rb_xor_agg(roaringbitmap)

描述：将分组内的roaringbitmap按照异或的逻辑组合成一个roaringbitmap。

示例：

```
SELECT a, rb_to_array(rb_xor_agg(b)) FROM r1 GROUP BY a ORDER BY a;
a   | rb_to_array
----+-------------
 1  | {}
 2  | {5}
 3  | {9}
 4  | {13}
 5  | {17}
 6  | {1}
 7  | {5}
 8  | {9}
 9  | {13}
 10 | {17}
(10 rows)
```

rb_and_cardinality_agg(roaringbitmap)

描述：分组内的roaringbitmap按照交集计算后的基数。

返回值类型：int

示例：

```
SELECT a, rb_and_cardinality_agg(b) FROM r1 GROUP BY a ORDER BY 1;
 a  | rb_and_cardinality_agg
----+------------------------
 1  |                      1
 2  |                      1
 3  |                      1
 4  |                      1
 5  |                      1
 6  |                      1
 7  |                      1
 8  |                      1
 9  |                      1
 10 |                      1
(10 rows)
```

rb_or_cardinality_agg(roaringbitmap)

描述：将分组内的roaringbitmap按照并集计算后的基数。

示例：

```
SELECT a, rb_or_cardinality_agg(b) FROM r1 GROUP BY a ORDER BY 1;
 a  | rb_or_cardinality_agg
----+-----------------------
 1  |                     1
 2  |                     2
 3  |                     2
 4  |                     2
 5  |                     2
 6  |                     2
 7  |                     2
 8  |                     2
 9  |                     2
 10 |                     2
(10 rows)
```

rb_xor_cardinality_agg(roaringbitmap)

描述：将分组内的roaringbitmap按照异或的逻辑合并后的基数。

示例：

```
SELECT a, rb_xor_cardinality_agg(b) FROM r1 GROUP BY a ORDER BY 1;
 a  | rb_xor_cardinality_agg
----+------------------------
 1  |                      0
 2  |                      1
 3  |                      1
 4  |                      1
 5  |                      1
 6  |                      1
 7  |                      1
 8  |                      1
 9  |                      1
 10 |                      1
(10 rows)
```

6.19.4 使用场景

背景

目前在互联网、教育、游戏等行业都有实时精准营销的需求。通过系统生成用户画像，在营销时通过条件组合筛选用户，快速提取目标群体。例如：

- 在电商行业中，商家在进行营销活动前，需要根据活动的目的，圈选一批满足特定特征的目标用户群体进行广告推送。
- 在教育行业中，需要根据学生不同的特征，推送有针对性的练习题目，帮助学生查漏补缺。
- 在搜索、视频、门户网站中，根据用户关注的热点，推送不同的内容。

这些业务场景都有一些共同的特点：

- 数据量庞大，运算量极大。
- 用户规模庞大，标签多，字段多，占用存储空间也多。
- 圈选的特征条件多样化，很难找到固定索引，如果每个字段一个索引，存储空间又会暴增。

- 性能要求高，因为实时营销要求秒级响应。
- 数据更新时效要求高，用户画像几乎要求实时更新。

针对上述业务场景特点，GaussDB(DWS)的roaringbitmap可以高效生成、压缩、解析位图数据，支持最常见的位图聚合操作（与、或、非、异或），满足用户在亿级以上、千万级标签的大数据量下实时精准营销、快速圈选用户的需求。

roaringbitmap 使用示例

假设有一张用户浏览网页的流水信息表userinfo，表中的字段如下：

```
CREATE TABLE userinfo
(userid int,
age int,
gender text,
salary int,
hobby  text
)WITH (orientation=column);
```

userinfo表中的数据会随着用户信息的变化不断增长，比如用户有多个hobby属性，那么就有多条记录。

如果用户需要筛选出所有“收入大于10000元的男性，年龄大于30岁，爱好钓鱼”的群体，向这些目标群体推送特定的消息。

传统的方法是直接在原表上执行查询，语句如下：

```
SELECT distinct userid FROM userinfo WHERE salary > 10000 AND age > 30 AND gender ='m' AND hobby ='fishing';
```

当userinfo表的数据量不大的时候，可以通过在salary, age, gender，hobby列上建立索引来满足需求。但是如果userinfo表的数据量非常大，同时一张表的标签数非常多的时候，上述语句就不能满足诉求，因为如下原因：

- 需要创建的索引会非常多。
- count (distinct)的性能比较差。

这种场景下使用roaringbitmap就会有比较好的效果。

1. 新建一张Roaringbitmap表：
   ```
   CREATE TABLE userinfoset
   ( age int,
   gender text,
   salary int,
   hobby  text,
   userset roaringbitmap,
   PRIMARY KEY(age,gender,salary,hobby)
   )WITH (orientation=column);
   ```
2. 所有userinfo表中的数据要通过标签列聚合到userinfoset表中。可以采用对全量数据进行聚合的方法（如下命令所示）。也可以采用只对增量数据进行聚合的方法。只对增量数据进行聚合即对含有相同的标签的用户集合放到表的一条记录中，通常可以通过upsert来实现。考虑到其中频繁的update操作可能产生大量的脏数据，因此对增量数据进行聚合的方法，建议将userinfoset表创建为行存表。
   ```
   INSERT INTO userinfoset
   SELECT age, gender, salary, hobby, rb_build_agg(userid)
   FROM
   userinfo
   GROUP BY age, gender, salary, hobby;
   ```
3. 直接查询userinfoset表获得用户筛选信息。
   ```
   SELECT rb_iterate(rb_or_agg(userset)) FROM userinfoset WHERE salary > 10000 AND age > 30 AND gender ='m' AND hobby ='fishing';
   ```


数据进行聚合后的userinfoset的数据量相比源表小了很多，基表scan的性能会快很多，同时基于Roaringbitmap的优势，计算rb_or_agg和rb_iterate的性能也很好，相比传统的方法，性能明显提升。

6.20 UUID 函数

UUID函数表示可以用于生成UUID类型（请参考**UUID类型**）数据的函数。

uuid_generate_v1()

描述：生成一个UUID类型的序列号。

返回类型：UUID

示例：

```
SELECT uuid_generate_v1();
          uuid_generate_v1
--------------------------------------
 c71ceaca-a175-11e9-a920-797ff7000001
(1 row)
```

说明

uuid_generate_v1函数根据时间信息、集群节点编号和生成该序列的线程号生成UUID，该UUID在单个集群内是全局唯一的，但在多个集群间的时间信息、集群节点编号、线程号和时钟序列仍然存在同时相等的可能性，因此多个集群间生成的UUID仍然存在极低概率的重复风险。

sys_guid()

描述：生成Oracle的GUID序列号，类似UUID。此函数为Oracle兼容性函数。

返回类型：text

示例：

```
SELECT sys_guid();
            sys_guid
----------------------------------
 4EBD3C74A17A11E9A1BF797FF7000001
(1 row)
```

说明

sys_guid函数内部生成原理同uuid_generate_v1函数。

UUID 函数应用示例

UUID全局唯一的特点，可以作为数据表生成主键，也可以作为数据表的分布列，uuid_generate_v1()作为数据表分布列的默认值时，通过Hash分布可以将数据均匀分布到各个DN上，防止数据倾斜。

说明

UUID的显著优点就是全局唯一，不需要中心节点，单个节点独立生成。但是也存在缺点，UUID较INT占用更多的存储空间，索引效率低，生成的ID随机，没有递增的特性，所以辨识困难。因此，在应用中，要根据实际情况选择UUID还是Sequence作为数据表主键。

示例如下：

1. INT类型作为分布列。

   创建示例哈希表mytable01，int类型作为分布列，插入数据后，查询数据存在数据倾斜。

```
CREATE TABLE mytable01(a INT, b INT) DISTRIBUTE BY hash(a);
CREATE TABLE
INSERT INTO mytable01 VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable01 VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable01 VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable01 VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable01 VALUES(1, 10);
INSERT 0 1
SELECT * FROM mytable01;
 a | b
---+----
 1 | 10
 1 | 10
 1 | 10
 1 | 10
 1 | 10
(5 rows)

SELECT table_skewness('mytable01');
          table_skewness
---------------------------------------
 ("dn_6003_6004        ",5,100.000%)
 ("dn_6001_6002        ",0,0.000%)
 ("dn_6005_6006        ",0,0.000%)
(3 rows)
```

2. UUID类型作为分布列。

   创建示例哈希表mytable02，UUID类型作为分布列，插入数据后，查询数据分布正常。

```
CREATE TABLE mytable02 (id UUID default uuid_generate_v1(), a INT, b INT) DISTRIBUTE BY hash(id);
CREATE TABLE

INSERT INTO mytable02(a, b) VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable02(a, b) VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable02(a, b) VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable02(a, b) VALUES(1, 10);
INSERT 0 1
INSERT INTO mytable02(a, b) VALUES(1, 10);
INSERT 0 1

SELECT * FROM mytable02;
                  id                  | a | b
--------------------------------------+---+----
 63e45c14-cc74-0e00-e9aa-0a2c3fa0fffe | 1 | 10
 63e45c1f-4d18-0700-e9ab-0a2c3fa0fffe | 1 | 10
 63e45c26-f859-0b00-e9ad-0a2c3fa0fffe | 1 | 10
 63e45c23-9e5d-0300-e9ac-0a2c3fa0fffe | 1 | 10
 63e45c2a-5825-0600-e9ae-0a2c3fa0fffe | 1 | 10
(5 rows)

SELECT table_skewness('mytable02');
          table_skewness
---------------------------------------
 ("dn_6001_6002        ",3,60.000%)
 ("dn_6003_6004        ",2,40.000%)
 ("dn_6005_6006        ",0,0.000%)
(3 rows)
```

6.21 文本检索函数和操作符

6.21.1 文本检索操作符

@@

描述：tsvector类型的词汇与tsquery类型的词汇是否匹配

示例：

```
SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat') AS RESULT;
 result
--------
 t
(1 row)
```

@@@

描述：@@的同义词

示例：

```
SELECT to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat') AS RESULT;
 result
--------
 t
(1 row)
```

&&

描述：将两个tsquery类型的词汇进行“与”操作

示例：

```
SELECT 'fat | rat'::tsquery && 'cat'::tsquery AS RESULT;
          result
---------------------------
 ( 'fat' | 'rat' ) & 'cat'
(1 row)
```

||

描述：将两个tsquery类型的词汇进行“或”操作

示例：

```
SELECT 'fat | rat'::tsquery || 'cat'::tsquery AS RESULT;
          result
---------------------------
 ( 'fat' | 'rat' ) | 'cat'
(1 row)
SELECT 'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector AS RESULT;
          result
----------------------------
 'a':1 'b':2,5 'c':3 'd':4
(1 row)
```


!!

描述：tsquery类型词汇的非关系

示例：

```
SELECT !! 'cat'::tsquery AS RESULT;
 result
--------
 !'cat'
(1 row)
```

@>

描述：一个tsquery类型的词汇是否包含另一个tsquery类型的词汇

示例：

```
SELECT 'cat'::tsquery @> 'cat & rat'::tsquery AS RESULT;
 result
--------
 t
(1 row)
```

<@

描述：一个tsquery类型的词汇是否被包含另一个tsquery类型的词汇

示例：

```
SELECT 'cat'::tsquery <@ 'cat & rat'::tsquery AS RESULT;
 result
--------
 t
(1 row)
```

除了上述的操作符，还为tsvector类型和tsquery类型的数据定义了普通的B-tree比较操作符（=，<等）。

6.21.2 文本检索函数

get_current_ts_config()

描述：获取文本检索的默认配置。

返回类型：regconfig

示例：

```
SELECT get_current_ts_config();
 get_current_ts_config
-----------------------
 english
(1 row)
```

length(tsvector)

描述：tsvector类型词汇的单词数。

返回类型：integer

示例：


```
SELECT length('fat:2,4 cat:3 rat:5A'::tsvector);
 length
--------
      3
(1 row)
```

numnode(tsquery)

描述：tsquery类型的单词加上操作符的数量。

返回类型：integer

示例：

```
SELECT numnode('(fat & rat) | cat'::tsquery);
 numnode
---------
       5
(1 row)
```

plainto_tsquery([config regconfig ,] query text)

描述：产生tsquery类型的词汇，并忽略标点。

返回类型：tsquery

示例：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
(1 row)
```

querytree(query tsquery)

描述：获取tsquery类型的词汇可加索引的部分。

返回类型：text

示例：

```
SELECT querytree('foo & ! bar'::tsquery);
 querytree
-----------
 'foo'
(1 row)
```

setweight(tsvector, "char")

描述：给tsvector类型的每个元素分配权值。

返回类型：tsvector

示例：

```
SELECT setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A');
           setweight
--------------------------------
 'cat':3A 'fat':2A,4A 'rat':5A
(1 row)
```


strip(tsvector)

描述：删除tsvector类型单词中的position和权值。

返回类型：tsvector

示例：

```
SELECT strip('fat:2,4 cat:3 rat:5A'::tsvector);
       strip
--------------------
 'cat' 'fat' 'rat'
(1 row)
```

to_tsquery([config regconfig ,] query text)

描述：标准化单词，并转换为tsquery类型。

返回类型：tsquery

示例：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
(1 row)
```

to_tsvector([config regconfig ,] document text)

描述：去除文件信息，并转换为tsvector类型。

返回类型：tsvector

示例：

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
(1 row)
```

ts_headline([config regconfig,] document text, query tsquery [, options text])

描述：高亮显示查询的匹配项。

返回类型：text

示例：

```
SELECT ts_headline('x y z', 'z'::tsquery);
 ts_headline
--------------
 x y <b>z</b>
(1 row)
```

ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer])

描述：文档查询排名。

返回类型：float4

示例：

```
SELECT ts_rank('hello world'::tsvector, 'world'::tsquery);
 ts_rank
----------
 .0607927
(1 row)
```

ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer])

描述：排序文件查询使用覆盖密度。

返回类型：float4

示例：

```
SELECT ts_rank_cd('hello world'::tsvector, 'world'::tsquery);
 ts_rank_cd
------------
          0
(1 row)
```

ts_rewrite(query tsquery, target tsquery, substitute tsquery)

描述：替换目标tsquery类型的单词。

返回类型：tsquery

示例：

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery);
        ts_rewrite
--------------------------
 'b' & ( 'foo' | 'bar' )
(1 row)
```

ts_rewrite(query tsquery, select text)

描述：使用SELECT命令的结果替代目标中tsquery类型的单词。

返回类型：tsquery

示例：

```
SELECT ts_rewrite('world'::tsquery, 'select ''world''::tsquery, ''hello''::tsquery');
 ts_rewrite
------------
 'hello'
(1 row)
```

6.21.3 文本检索调试函数

ts_debug([config regconfig,] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])

描述：测试一个配置。

返回类型：setof record

示例：


```
SELECT ts_debug('english', 'The Brightest supernovaes');
                                   ts_debug
-------------------------------------------------------------------------------
 (asciiword,"Word, all ASCII",The,{english_stem},english_stem,{})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all ASCII",Brightest,{english_stem},english_stem,{brightest})
 (blank,"Space symbols"," ",{},,)
 (asciiword,"Word, all ASCII",supernovaes,{english_stem},english_stem,{supernova})
(5 rows)
```

ts_lexize(dict regdictionary, token text)

描述：测试一个数据字典。

返回类型：text[]

示例：

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}
(1 row)
```

ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)

描述：测试一个解析。

返回类型：setof record

示例：

```
SELECT ts_parse('default', 'foo - bar');
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
(4 rows)
```

ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)

描述：测试一个解析。

返回类型：setof record

示例：

```
SELECT ts_parse(3722, 'foo - bar');
 ts_parse
-----------
 (1,foo)
 (12," ")
 (12,"- ")
 (1,bar)
(4 rows)
```

ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)

描述：获取分析器定义的记号类型。

返回类型：setof record

示例：

```
SELECT ts_token_type('default');
                      ts_token_type
---------------------------------------------------------------
 (1,asciiword,"Word, all ASCII")
 (2,word,"Word, all letters")
 (3,numword,"Word, letters and digits")
 (4,email,"Email address")
 (5,url,URL)
 (6,host,Host)
 (7,sfloat,"Scientific notation")
 (8,version,"Version number")
 (9,hword_numpart,"Hyphenated word part, letters and digits")
 (10,hword_part,"Hyphenated word part, all letters")
 (11,hword_asciipart,"Hyphenated word part, all ASCII")
 (12,blank,"Space symbols")
 (13,tag,"XML tag")
 (14,protocol,"Protocol head")
 (15,numhword,"Hyphenated word, letters and digits")
 (16,asciihword,"Hyphenated word, all ASCII")
 (17,hword,"Hyphenated word, all letters")
 (18,url_path,"URL path")
 (19,file,"File or path name")
 (20,float,"Decimal notation")
 (21,int,"Signed integer")
 (22,uint,"Unsigned integer")
 (23,entity,"XML entity")
(23 rows)
```

ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)

描述：获取分析器定义的记号类型。

返回类型：setof record

示例：

```
SELECT ts_token_type(3722);
                      ts_token_type
---------------------------------------------------------------
 (1,asciiword,"Word, all ASCII")
 (2,word,"Word, all letters")
 (3,numword,"Word, letters and digits")
 (4,email,"Email address")
 (5,url,URL)
 (6,host,Host)
 (7,sfloat,"Scientific notation")
 (8,version,"Version number")
 (9,hword_numpart,"Hyphenated word part, letters and digits")
 (10,hword_part,"Hyphenated word part, all letters")
 (11,hword_asciipart,"Hyphenated word part, all ASCII")
 (12,blank,"Space symbols")
 (13,tag,"XML tag")
 (14,protocol,"Protocol head")
 (15,numhword,"Hyphenated word, letters and digits")
 (16,asciihword,"Hyphenated word, all ASCII")
 (17,hword,"Hyphenated word, all letters")
 (18,url_path,"URL path")
 (19,file,"File or path name")
 (20,float,"Decimal notation")
 (21,int,"Signed integer")
 (22,uint,"Unsigned integer")
 (23,entity,"XML entity")
(23 rows)
```


ts_stat(sqlquery text, [weights text,] OUT word text, OUT ndoc integer, OUT nentry integer)

描述：获取tsvector列的统计数据。

返回类型：setof record

示例：

```
SELECT ts_stat('select ''hello world''::tsvector');
   ts_stat
-------------
 (world,1,1)
 (hello,1,1)
(2 rows)
```

6.22 HLL 函数和操作符

6.22.1 HLL 操作符

HLL类型支持如下操作符：

表 6-17 HLL 操作符

HLL操作符	描述	返回值类型	示例
=	比较hll或hll_hashval的值是否相等。	bool	• hll SELECT (hll_empty() || hll_hash_integer(1)) = (hll_empty() || hll_hash_integer(1)); ?column? ---------- t (1 row) • hll_hashval SELECT hll_hash_integer(1) = hll_hash_integer(1); ?column? ---------- t (1 row)
<> or !=	比较hll或hll_hashval是否不相等。	bool	• hll SELECT (hll_empty() || hll_hash_integer(1)) <> (hll_empty() || hll_hash_integer(2)); ?column? ---------- t (1 row) • hll_hashval SELECT hll_hash_integer(1) <> hll_hash_integer(2); ?column? ---------- t (1 row)

HLL操作符	描述	返回值类型	示例
||	可代表hll_add, hll_union, hll_add_rev三个函数的功能。	hll	• hll_add SELECT hll_empty() || hll_hash_integer(1); ?column? ---------- \x128b7f8895a3f5af28cafe (1 row) • hll_add_rev SELECT hll_hash_integer(1) || hll_empty(); ?column? ---------- \x128b7f8895a3f5af28cafe (1 row) • hll_union SELECT (hll_empty() || hll_hash_integer(1)) || (hll_empty() || hll_hash_integer(2)); ?column? ---------- \x128b7f8895a3f5af28cafeda0ce907e4355b60 (1 row)
#	计算出hll的Dintinct值, 同hll_cardinality函数。	integer	SELECT #(hll_empty() || hll_hash_integer(1)); ?column? ---------- 1 (1 row)

6.22.2 哈希函数

hll_hash_boolean(bool)

描述：对bool类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_boolean(FALSE);
  hll_hash_boolean
---------------------
 5048724184180415669
(1 row)
```

hll_hash_boolean(bool, int32)

描述：设置hash seed（即改变哈希策略）并对bool类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_boolean(FALSE, 10);
  hll_hash_boolean
--------------------
 391264977436098630
(1 row)
```

hll_hash_smallint(smallint)

描述：对smallint类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_smallint(100::smallint);
  hll_hash_smallint
---------------------
 4631120266694327276
(1 row)
```

说明

数值大小相同的参数使用不同数据类型的哈希函数计算，最后结果会不一样，因为不同类型哈希函数会选取不同的哈希计算策略。

hll_hash_smallint(smallint, int32)

描述：设置hash seed（即改变哈希策略）同时对smallint类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_smallint(100::smallint, 10);
  hll_hash_smallint
---------------------
 8349353095166695771
(1 row)
```

hll_hash_integer(integer)

描述：对integer类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_integer(0);
   hll_hash_integer
----------------------
 -3485513579396041028
(1 row)
```

hll_hash_integer(integer, int32)

描述：对integer类型数据计算哈希值，并设置hashseed（即改变哈希策略）。

返回值类型：hll_hashval

示例：

```
 SELECT hll_hash_integer(0, 10);
  hll_hash_integer
---------------------
 183371090322255134
(1 row)
```

hll_hash_bigint(bigint)

描述：对bigint类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_bigint(100::bigint);
   hll_hash_bigint
---------------------
 8349353095166695771
(1 row)
```

hll_hash_bigint(bigint, int32)

描述：对bigint类型数据计算哈希值，并设置hashseed（即改变哈希策略）。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_bigint(100::bigint, 10);
   hll_hash_bigint
---------------------
 4631120266694327276
(1 row)
```

hll_hash_bytea(bytea)

描述：对bytea类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_bytea(E'\\x');
 hll_hash_bytea
----------------
 0
(1 row)
```

hll_hash_bytea(bytea, int32)

描述：对bytea类型数据计算哈希值，并设置hashseed（即改变哈希策略）。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_bytea(E'\\x', 10);
   hll_hash_bytea
---------------------
 6574525721897061910
(1 row)
```

hll_hash_text(text)

描述：对text类型数据计算哈希值。

返回值类型：hll_hashval

示例：


```
SELECT hll_hash_text('AB');
    hll_hash_text
---------------------
 5365230931951287672
(1 row)
```

hll_hash_text(text, int32)

描述：对text类型数据计算哈希值, 并设置hashseed（即改变哈希策略）。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_text('AB', 10);
    hll_hash_text
---------------------
 7680762839921155903
(1 row)
```

hll_hash_any(anytype)

描述：对任意类型数据计算哈希值。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_any(1);
     hll_hash_any
----------------------
 -8604791237420463362
(1 row)

SELECT hll_hash_any('08:00:2b:01:02:03'::macaddr);
     hll_hash_any
----------------------
 -4883882473551067169
(1 row)
```

hll_hash_any(anytype, int32)

描述：对任意类型数据计算哈希值，并设置hashseed（即改变哈希策略）。

返回值类型：hll_hashval

示例：

```
SELECT hll_hash_any(1, 10);
     hll_hash_any
----------------------
 -1478847531811254870
(1 row)
```

hll_hashval_eq(hll_hashval, hll_hashval)

描述：比较两个hll_hashval类型数据是否相等。

返回值类型：bool

示例：

```
SELECT hll_hashval_eq(hll_hash_integer(1), hll_hash_integer(1));
 hll_hashval_eq
```



```
----------------
 t
(1 row)
```

hll_hashval_ne(hll_hashval, hll_hashval)

描述：比较两个hll_hashval类型数据是否不相等。

返回值类型：bool

示例：

```
SELECT hll_hashval_ne(hll_hash_integer(1), hll_hash_integer(1));
 hll_hashval_ne
----------------
 f
(1 row)
```

6.22.3 精度函数

HLL（HyperLogLog）主要存在三种模式Explicit，Sparse，Full。当数据规模比较小的时候会使用Explicit模式和Sparse模式， 这两种模式在计算结果上基本上没有误差。随着distinct值越来越多，就会转换成Full模式，但结果也会存在一定误差。下列函数用于查看HLL中精度参数。

hll_schema_version(hll)

描述：查看当前hll中的schema version。

返回值类型：integer

示例：

```
SELECT hll_schema_version(hll_empty());
 hll_schema_version
--------------------
                  1
(1 row)
```

hll_type(hll)

描述：查看当前hll的类型。

返回值类型：integer

示例：

```
SELECT hll_type(hll_empty());
 hll_type
----------
        1
(1 row)
```

hll_log2m(hll)

描述：查看当前hll的log2m数值，此值会影响最后hll计算distinct误差率，误差率计算公式为：

$$\pm 1.04/\sqrt{2 \wedge log2m}$$

返回值类型：integer

示例：

```
SELECT hll_log2m(hll_empty());
 hll_log2m
-----------
        11
(1 row)
```

hll_regwidth(hll)

描述：查看hll数据结构中桶的位数大小。

返回值类型：integer

示例：

```
SELECT hll_regwidth(hll_empty());
 hll_regwidth
--------------
           5
(1 row)
```

hll_expthresh(hll)

描述：得到当前hll中expthresh大小，hll通常会由Explicit模式到Sparse模式再到Full模式，这个过程称为promotion hierarchy策略。可以通过调整expthresh值的大小改变策略，比如expthresh为0的时候就会跳过Explicit模式而直接进入Sparse模式。当显式指定expthresh的取值为1-7之间时，该函数得到的是 $2^{expthresh}$。

返回值类型：record

示例：

```
SELECT hll_expthresh(hll_empty());
 hll_expthresh
---------------
 (-1,160)
(1 row)

SELECT hll_expthresh(hll_empty(11,5,3));
 hll_expthresh
---------------
 (8,8)
(1 row)
```

hll_sparseon(hll)

描述：是否启用sparse模式，0是关闭，1是开启。

返回值类型：integer

示例：

```
SELECT hll_sparseon(hll_empty());
 hll_sparseon
--------------
           1
(1 row)
```


6.22.4 聚合函数

hll_add_agg(hll_hashval)

描述：把哈希后的数据按照分组放到hll中。

返回值类型：hll

示例：

1. 准备数据。
```
CREATE TABLE t_id(id int);
INSERT INTO t_id VALUES(generate_series(1,500));
CREATE TABLE t_data(a int, c text);
INSERT INTO t_data SELECT mod(id,2), id FROM t_id;
```
2. 创建表并指定列为hll。
```
CREATE TABLE t_a_c_hll(a int, c hll);
```
3. 根据a列group by对数据分组，把各组数据加到hll中。
```
INSERT INTO t_a_c_hll SELECT a, hll_add_agg(hll_hash_text(c)) FROM t_data GROUP BY a;
```
4. 得到每组数据中hll的Distinct值。
```
SELECT a, #c as cardinality FROM t_a_c_hll order by a;
 a |   cardinality
---+------------------
 0 | 250.741759091658
 1 | 250.741759091658
(2 rows)
```

hll_add_agg(hll_hashval, int32 log2m)

描述：把哈希后的数据按照分组放到hll中。 并指定参数log2m，取值范围为10~16。

返回值类型：hll

示例：

```
 SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), 10)) FROM t_data;
 hll_cardinality
------------------
 503.932348927339
(1 row)
```

hll_add_agg(hll_hashval, int32 log2m, int32 regwidth)

描述：把哈希后的数据按照分组放到hll中。依次制定参数log2m， regwidth。regwidth取值范围为1~5。

返回值类型：hll

示例：

```
SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1)) FROM t_data;
 hll_cardinality
------------------
 496.628982624022
(1 row)
```

hll_add_agg(hll_hashval, int32 log2m, int32 regwidth, int64 expthresh)

描述：把哈希后的数据按照分组放到hll中, 依次指定参数log2m、regwidth、expthresh。expthresh的取值范围是-1~7之间的整数，该参数可以用来设置从Explicit

模式到Sparse模式的阈值大小。-1表示自动模式，0表示跳过Explicit模式，取1~7表示在基数到达 $2^{expthresh}$时切换模式。

返回值类型：hll

示例：

```
 SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1, 4)) FROM t_data;
 hll_cardinality
------------------
 496.628982624022
(1 row)
```

hll_add_agg(hll_hashval, int32 log2m, int32 regwidth, int64 expthresh, int32 sparseon)

描述：把哈希后的数据按照分组放到hll中, 依次制定参数log2m、regwidth、expthresh、sparseon。sparseon取值范围为0或者1。

返回值类型：hll

示例：

```
 SELECT hll_cardinality(hll_add_agg(hll_hash_text(c), NULL, 1, 4, 0)) FROM t_data;
 hll_cardinality
------------------
 496.628982624022
(1 row)
```

hll_union_agg(hll)

描述：将多个hll类型数据union成一个hll。

返回值类型：hll

示例：

将各组中的hll数据union成一个hll，并计算distinct值。

```
SELECT #hll_union_agg(c) as cardinality FROM t_a_c_hll;
   cardinality
------------------
 496.628982624022
(1 row)
```

说明

当两个或者多个hll数据结构执行union时，必须要保证其中每一个hll里面的精度参数一样，否则不能进行union。同样的约束也适用于函数hll_union(hll,hll)。

6.22.5 功能函数

hll_print(hll)

描述：打印hll的一些debug参数信息。

返回值类型：cstring

示例：

```
SELECT hll_print(hll_empty());
                    hll_print
```



```
-------------------------------------------------------------
 EMPTY, nregs=2048, nbits=5, expthresh=-1(160), sparseon=1gongne
(1 row)
```

hll_empty()

描述：创建一个空的hll。

返回值类型：hll

示例：

```
SELECT hll_empty();
 hll_empty
-----------
 \x118b7f
(1 row)
```

hll_empty(int32 log2m)

描述：创建空的hll并指定参数log2m，取值范围是10到16。

返回值类型: hll

示例：

```
SELECT hll_empty(10);
 hll_empty
-----------
 \x118a7f
(1 row)
```

hll_empty(int32 log2m, int32 regwidth)

描述：创建空的hll并依次指定参数log2m、regwidth。regwidth取值范围是1到5。

返回值类型: hll

示例：

```
SELECT hll_empty(10, 4);
 hll_empty
-----------
 \x116a7f
(1 row)
```

hll_empty(int32 log2m, int32 regwidth, int64 expthresh)

描述：创建空的hll并依次指定参数log2m、regwidth、expthresh。expthresh取值范围是-1到7之间的整数。该参数可以用来设置从Explicit模式到Sparse模式的阈值大小。-1表示自动模式，0表示跳过Explicit模式，取1-7表示在基数到达$2^{expthresh}$时切换模式。

返回值类型：hll

示例：

```
SELECT hll_empty(10, 4, 7);
 hll_empty
-----------
 \x116a48
(1 row)
```


hll_empty(int32 log2m, int32 regwidth, int64 expthresh, int32 sparseon)

描述：创建空的hll并依次指定参数log2m、regwidth、expthresh、sparseon。sparseon取0或者1。

返回值类型：hll

示例：

```
SELECT hll_empty(10,4,7,0);
 hll_empty
-----------
 \x116a08
(1 row)
```

hll_add(hll, hll_hashval)

描述：把hll_hashval加入到hll中。

返回值类型：hll

示例：

```
SELECT hll_add(hll_empty(), hll_hash_integer(1));
          hll_add
--------------------------
 \x128b7f8895a3f5af28cafe
(1 row)
```

hll_add_rev(hll_hashval, hll)

描述：把hll_hashval加入到hll中，和hll_add功能一样，只是参数位置进行了交换。

返回值类型：hll

示例：

```
SELECT hll_add_rev(hll_hash_integer(1), hll_empty());
       hll_add_rev
--------------------------
 \x128b7f8895a3f5af28cafe
(1 row)
```

hll_eq(hll, hll)

描述：比较两个hll是否相等。

返回值类型：bool

示例：

```
SELECT hll_eq(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
 hll_eq
--------
 f
(1 row)
```

hll_ne(hll, hll)

描述：比较两个hll是否不相等。

返回值类型：bool

示例：

```
SELECT hll_ne(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
 hll_ne
--------
 t
(1 row)
```

hll_cardinality(hll)

描述：计算hll的distinct值。

返回值类型：integer

示例：

```
SELECT hll_cardinality(hll_empty() || hll_hash_integer(1));
 hll_cardinality
-----------------
               1
(1 row)
```

hll_union(hll, hll)

描述：把两个hll数据结构union成一个。

返回值类型：hll

示例：

```
SELECT hll_union(hll_add(hll_empty(), hll_hash_integer(1)), hll_add(hll_empty(), hll_hash_integer(2)));
                                hll_union
-----------------------------------------------------------
 \x128b7f8895a3f5af28cafeda0ce907e4355b60
(1 row)
```

6.22.6 内置函数

HLL（HyperLogLog）有一系列内置函数用于内部对数据进行处理，一般情况下不建议用户使用。

表 6-18 内置函数

函数名称	功能描述
hll_in	以string格式接收hll数据。
hll_out	以string格式发送hll数据。
hll_recv	以bytea格式接收hll数据。
hll_send	以bytea格式发送hll数据。
hll_trans_in	以string格式接收hll_trans_type数据。
hll_trans_out	以string格式发送hll_trans_type数据。
hll_trans_recv	以bytea形式接收hll_trans_type数据。
hll_trans_send	以bytea形式发送hll_trans_type数据。

函数名称	功能描述
hll_typmod_in	接收typmod类型数据。
hll_typmod_out	发送typmod类型数据。
hll_hashval_in	接收hll_hashval类型数据。
hll_hashval_out	发送hll_hashval类型数据。
hll_add_trans0	类似于hll_add所提供的功能, 通常在分布式聚合运算的第一阶段DN上使用。
hll_union_trans	类似hll_union所提供的功能，在分布式聚合运算的第一阶段DN上使用。
hll_union_collect	类似于hll_union所提供的功能，在分布式聚合运算第二阶段CN上使用，汇总各个DN上的结果。
hll_pack	在分布式聚合运算第三阶段CN上使用，把自定义hll_trans_type类型最后转换成hll类型。
hll	用于hll类型转换成hll类型，根据输入参数会设定指定参数。
hll_hashval	用于bigint类型转换成hll_hashval类型。
hll_hashval_int4	用于int4类型转换成hll_hashval类型。

6.23 集合返回函数

6.23.1 序列号生成函数

generate_series()函数根据指定的开始值(start)、结束值(stop)和步长(step)返回一个基于系列的集合。

generate_series()函数的入参中，当step是正数且start大于stop，则返回零行。相反，当step是负数且start小于stop，则返回零行。如果任何输入为NULL也会返回零行。如果step为零则会报错。

generate_series(start, stop)

描述：生成一个数值序列，从start到stop，步长默认为1。

参数类型：int、bigint、numeric

返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

示例：

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)
```

```
SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT * FROM generate_series(1,NULL);
 generate_series
-----------------
(0 rows)
```

generate_series(start, stop, step)

描述：生成一个数值序列，从start到stop，步长为step。

参数类型：int、bigint、numeric

返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

示例：

```
SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,6,-5);
 generate_series
-----------------
(0 rows)

SELECT * FROM generate_series(4,3,0);
ERROR:  step size cannot equal zero
```

generate_series(start, stop, step interval)

描述：生成一个数值序列，从start到stop，步长为step。

参数类型：timestamp或timestamp with time zone

返回值类型：setof timestamp或setof timestamp with time zone（与参数类型相同）

示例：

```
--这个示例应用于date-plus-integer操作符。
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2017-06-02
 2017-06-09
 2017-06-16
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp, '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
```



```
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

6.23.2 下标生成函数

generate_subscripts(array anyarray, dim int)

描述：生成一系列包括给定数组的下标。

返回值类型：setof int

示例：

```
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)
```

generate_subscripts(array anyarray, dim int, reverse boolean)

描述：生成一系列包括给定数组的下标。当reverse为真时，该系列则以相反的顺序返回。

返回值类型：setof int

示例：

```
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1,TRUE) AS s;
 s
---
 4
 3
 2
 1
(4 rows)
```

generate_subscripts是为给定数组的指定维度生成有效下标集的函数。如果数组中没有所请求的维度或者任何输入NULL数组，均返回零行（但是会给数组元素为空的返回有效下标）。示例：

```
--unnest一个2D数组。
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
SELECT $1[i][j]
   FROM generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;

SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```


6.24 几何函数和操作符

6.24.1 几何操作符

+

描述：平移，即从第一个参数的每个点的坐标中加上第二个point的坐标。

示例：

```
SELECT box '((0,0),(1,1))' + point '(2.0,0)' AS RESULT;
   result
-------------
 (3,1),(2,0)
(1 row)
```

-

描述：平移，从第一个参数的每个点的坐标中减去第二个point的坐标。

示例：

```
SELECT box '((0,0),(1,1))' - point '(2.0,0)' AS RESULT;
    result
---------------
 (-1,1),(-2,0)
(1 row)
```

*****

描述：伸展/旋转。

示例：

```
SELECT box '((0,0),(1,1))' * point '(2.0,0)' AS RESULT;
   result
-------------
 (2,2),(0,0)
(1 row)
```

/

描述：收缩/旋转。

示例：

```
SELECT box '((0,0),(2,2))' / point '(2.0,0)' AS RESULT;
   result
-------------
 (1,1),(0,0)
(1 row)
```

#

描述：两个图形交点或者交面。

示例：


```
SELECT box'((1,-1),(-1,1))' # box'((1,1),(-1,-1))' AS RESULT;
 result
--------
(1,1),(-1,-1)
(1 row)
```

#

描述：图形的路径数目或多边形顶点数。

示例：

```
SELECT # path'((1,0),(0,1),(-1,0))' AS RESULT;
 result
--------
      3
(1 row)
```

@-@

描述：图形的长度或者周长。

示例：

```
SELECT @-@ path '((0,0),(1,0))' AS RESULT;
 result
--------
      2
(1 row)
```

@@

描述：图形的中心。

示例：

```
SELECT @@ circle '((0,0),10)' AS RESULT;
 result
--------
 (0,0)
(1 row)
```

##

描述：第一个图形相对第二个图形的最近点。

示例：

```
SELECT point '(0,0)' ## box '((2,0),(0,2))' AS RESULT;
 result
--------
 (0,0)
(1 row)
```

<->

描述：两个图形之间的距离。

示例：

```
SELECT circle '((0,0),1)' <-> circle '((5,0),1)' AS RESULT;
 result
--------
```



```
    3
(1 row)
```

&&

描述：两个图形是否重叠（有一个共同点就为真）。

示例：

```
SELECT box '((0,0),(1,1))' && box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

<<

描述：图形是否全部在另一个图形的左边（没有相同的横坐标）。

示例：

```
SELECT circle '((0,0),1)' << circle '((5,0),1)' AS RESULT;
 result
--------
 t
(1 row)
```

>>

描述：图形是否全部在另一个图形的右边（没有相同的横坐标）。

示例：

```
SELECT circle '((5,0),1)' >> circle '((0,0),1)' AS RESULT;
 result
--------
 t
(1 row)
```

&<

描述：图形的最右边是否不超过在另一个图形的最右边。

示例：

```
SELECT box '((0,0),(1,1))' &< box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

&>

描述：图形的最左边是否不超过在另一个图形的最左边。

示例：

```
SELECT box '((0,0),(3,3))' &> box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```


<<|

描述：图形是否全部在另一个图形的下边（没有相同的纵坐标）。

示例：

```
SELECT box '((0,0),(3,3))' <<| box '((3,4),(5,5))' AS RESULT;
 result
--------
 t
(1 row)
```

|>>

描述：图形是否全部在另一个图形的上边（没有相同的纵坐标）。

示例：

```
SELECT box '((3,4),(5,5))' |>> box '((0,0),(3,3))' AS RESULT;
 result
--------
 t
(1 row)
```

&<|

描述：图形的最上边是否不超过另一个图形的最上边。

示例：

```
SELECT box '((0,0),(1,1))' &<| box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

|&>

描述：图形的最下边是否不超过另一个图形的最下边。

示例：

```
SELECT box '((0,0),(3,3))' |&> box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

<^

描述：图形是否低于另一个图形（允许两个图形有接触）。

示例：

```
SELECT box '((0,0),(-3,-3))' <^ box '((0,0),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

>^

描述：图形是否高于另一个图形（允许两个图形有接触）。

示例：

```
SELECT box '((0,0),(2,2))' >^ box '((0,0),(-3,-3))'  AS RESULT;
 result
--------
 t
(1 row)
```

?#

描述：两个图形是否相交。

示例：

```
SELECT lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))' AS RESULT;
 result
--------
 t
(1 row)
```

?-

描述：图形是否处于水平位置。

示例：

```
SELECT ?- lseg '((-1,0),(1,0))' AS RESULT;
 result
--------
 t
(1 row)
```

?-

描述：图形是否水平对齐。

示例：

```
SELECT point '(1,0)' ?- point '(0,0)' AS RESULT;
 result
--------
 t
(1 row)
```

?|

描述：图形是否处于竖直位置。

示例：

```
SELECT ?| lseg '((-1,0),(1,0))' AS RESULT;
 result
--------
 f
(1 row)
```

?|

描述：图形是否竖直对齐。

示例：

```
SELECT point '(0,1)' ?| point '(0,0)' AS RESULT;
 result
```



```
--------
 t
(1 row)
```

?-|

描述：两条线是否垂直。

示例：

```
SELECT lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))' AS RESULT;
 result
--------
 t
(1 row)
```

?||

描述：两条线是否平行。

示例：

```
SELECT lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))' AS RESULT;
 result
--------
 t
(1 row)
```

@>

描述：图形是否包含另一个图形。

示例：

```
SELECT circle '((0,0),2)' @> point '(1,1)' AS RESULT;
 result
--------
 t
(1 row)
```

<@

描述：图形是否被包含于另一个图形。

示例：

```
SELECT point '(1,1)' <@ circle '((0,0),2)' AS RESULT;
 result
--------
 t
(1 row)
```

~=

描述：两个图形是否相同?

示例：

```
SELECT polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' AS RESULT;
 result
--------
 t
(1 row)
```


6.24.2 几何函数

area(object)

描述：计算图形的面积。

返回类型：double precision

示例：

```
SELECT area(box '((0,0),(1,1))') AS RESULT;
 result
--------
      1
(1 row)
```

center(object)

描述：计算图形的中心。

返回类型：point

示例：

```
SELECT center(box '((0,0),(1,2))') AS RESULT;
 result
---------
 (0.5,1)
(1 row)
```

diameter(circle)

描述：计算圆的直径。

返回类型：double precision

示例：

```
SELECT diameter(circle '((0,0),2.0)') AS RESULT;
 result
--------
      4
(1 row)
```

height(box)

描述：矩形的竖直高度。

返回类型：double precision

示例：

```
SELECT height(box '((0,0),(1,1))') AS RESULT;
 result
--------
      1
(1 row)
```

isclosed(path)

描述：图形是否为闭合路径。

返回类型：boolean

示例：

```
SELECT isclosed(path '((0,0),(1,1),(2,0))') AS RESULT;
 result
--------
 t
(1 row)
```

isopen(path)

描述：图形是否为开放路径。

返回类型：boolean

示例：

```
SELECT isopen(path '[(0,0),(1,1),(2,0)]') AS RESULT;
 result
--------
 t
(1 row)
```

length(object)

描述：计算图形的长度。

返回类型：double precision

示例：

```
SELECT length(path '((-1,0),(1,0))') AS RESULT;
 result
--------
      4
(1 row)
```

npoints(path)

描述：计算路径的顶点数。

返回类型：int

示例：

```
SELECT npoints(path '[(0,0),(1,1),(2,0)]') AS RESULT;
 result
--------
      3
(1 row)
```

npoints(polygon)

描述：计算多边形的顶点数。

返回类型：int

示例：

```
SELECT npoints(polygon '((1,1),(0,0))') AS RESULT;
 result
--------
```



```
     2
(1 row)
```

pclose(path)

描述：把路径转换为闭合路径。

返回类型：path

示例：

```
SELECT pclose(path '[(0,0),(1,1),(2,0)]') AS RESULT;
       result
---------------------
 ((0,0),(1,1),(2,0))
(1 row)
```

popen(path)

描述：把路径转换为开放路径。

返回类型：path

示例：

```
SELECT popen(path '((0,0),(1,1),(2,0))') AS RESULT;
       result
---------------------
 [(0,0),(1,1),(2,0)]
(1 row)
```

radius(circle)

描述：计算圆的半径。

返回类型：double precision

示例：

```
SELECT radius(circle '((0,0),2.0)') AS RESULT;
 result
--------
     2
(1 row)
```

width(box)

描述：计算矩形的水平尺寸。

返回类型：double precision

示例：

```
SELECT width(box '((0,0),(1,1))') AS RESULT;
 result
--------
     1
(1 row)
```


6.24.3 几何类型转换函数

box(circle)

描述：将圆转换成矩形。

返回类型：box

示例：

```
SELECT box(circle '((0,0),2.0)') AS RESULT;
                                result
-------------------------------------------------------------------------
 (1.41421356237309,1.41421356237309),(-1.41421356237309,-1.41421356237309)
(1 row)
```

box(point, point)

描述：将点转换成矩形。

返回类型：box

示例：

```
SELECT box(point '(0,0)', point '(1,1)') AS RESULT;
  result
-------------
 (1,1),(0,0)
(1 row)
```

box(polygon)

描述：将多边形转换成矩形。

返回类型：box

示例：

```
SELECT box(polygon '((0,0),(1,1),(2,0))') AS RESULT;
  result
-------------
 (2,1),(0,0)
(1 row)
```

circle(box)

描述：矩形转换成圆。

返回类型：circle

示例：

```
SELECT circle(box '((0,0),(1,1))') AS RESULT;
            result
-------------------------------
 <(0.5,0.5),0.707106781186548>
(1 row)
```

circle(point, double precision)

描述：将圆心和半径转换成圆。

返回类型：circle

示例：

```
SELECT circle(point '(0,0)', 2.0) AS RESULT;
  result
----------
 <(0,0),2>
(1 row)
```

circle(polygon)

描述：将多边形转换成圆。

返回类型：circle

示例：

```
SELECT circle(polygon '((0,0),(1,1),(2,0))') AS RESULT;
                  result
-----------------------------------------------
 <(1,0.333333333333333),0.924950591148529>
(1 row)
```

lseg(box)

描述：矩形对角线转化成线段。

返回类型：lseg

示例：

```
SELECT lseg(box '((-1,0),(1,0))') AS RESULT;
    result
----------------
 [(1,0),(-1,0)]
(1 row)
```

lseg(point, point)

描述：点转换成线段。

返回类型：lseg

示例：

```
SELECT lseg(point '(-1,0)', point '(1,0)') AS RESULT;
    result
----------------
 [(-1,0),(1,0)]
(1 row)
```

path(polygon)

描述：多边形转换成路径。

返回类型：path

示例：

```
SELECT path(polygon '((0,0),(1,1),(2,0))') AS RESULT;
      result
--------------------
```



```
 ((0,0),(1,1),(2,0))
(1 row)
```

point(double precision, double precision)

描述：结点。

返回类型：point

示例：

```
SELECT point(23.4, -44.5) AS RESULT;
   result
--------------
 (23.4,-44.5)
(1 row)
```

point(box)

描述：矩形的中心。

返回类型：point

示例：

```
SELECT point(box '((-1,0),(1,0))') AS RESULT;
 result
--------
 (0,0)
(1 row)
```

point(circle)

描述：圆心。

返回类型：point

示例：

```
SELECT point(circle '((0,0),2.0)') AS RESULT;
 result
--------
 (0,0)
(1 row)
```

point(lseg)

描述：线段的中心。

返回类型：point

示例：

```
SELECT point(lseg '((-1,0),(1,0))') AS RESULT;
 result
--------
 (0,0)
(1 row)
```

point(polygon)

描述：多边形的中心。

返回类型：point

示例：

```
SELECT point(polygon '((0,0),(1,1),(2,0))') AS RESULT;
        result
-----------------------
 (1,0.333333333333333)
(1 row)
```

polygon(box)

描述：矩形转换成4点多边形。

返回类型：polygon

示例：

```
SELECT polygon(box '((0,0),(1,1))') AS RESULT;
          result
----------------------------
 ((0,0),(0,1),(1,1),(1,0))
(1 row)
```

polygon(circle)

描述：圆转换成12点多边形。

返回类型：polygon

示例：

```
SELECT polygon(circle '((0,0),2.0)') AS RESULT;

result

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 ((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),(1,1.73205080756888),
(1.73205080756888,1),(2,2.44929359829471e-16),(1.73205080756888,-0.999999999999999),
(1,-1.73205080756888),(3.67394039744206e-16,-2),(-0.999999999999999,-1.73205080756888),
(-1.73205080756888,-1))
(1 row)
```

polygon(npts, circle)

描述：圆转换成npts点多边形。

返回类型：polygon

示例：

```
SELECT polygon(12, circle '((0,0),2.0)') AS RESULT;

result

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 ((-2,0),(-1.73205080756888,1),(-1,1.73205080756888),(-1.22464679914735e-16,2),(1,1.73205080756888),
(1.73205080756888,1),(2,2.44929359829471e-16),(1.73205080756888,-0.999999999999999),
(1,-1.73205080756888),(3.67394039744206e-16,-2),(-0.999999999999999,-1.73205080756888),
(-1.73205080756888,-1))
(1 row)
```


polygon(path)

描述：路径转换成多边形。

返回类型：polygon

示例：

```
SELECT polygon(path '((0,0),(1,1),(2,0))') AS RESULT;
      result
---------------------
 ((0,0),(1,1),(2,0))
(1 row)
```

6.25 网络地址函数和操作符

6.25.1 cidr 和 inet 操作符

操作符<<，<<=，>>，>>=对子网包含进行测试。它们只考虑两个地址的网络部分（忽略任何主机部分），然后判断其中一个网络是等于另外一个网络，还是另外一个网络的子网。

<

描述：小于

示例：

```
SELECT inet '192.168.1.5' < inet '192.168.1.6' AS RESULT;
 result
--------
 t
(1 row)
```

<=

描述：小于或等于

示例：

```
SELECT inet '192.168.1.5' <= inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```

=

描述：等于

示例：

```
SELECT inet '192.168.1.5' = inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```


>=

描述：大于或等于

示例：

```
SELECT inet '192.168.1.5' >= inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```

>

描述：大于

示例：

```
SELECT inet '192.168.1.5' > inet '192.168.1.4' AS RESULT;
 result
--------
 t
(1 row)
```

<>

描述：不等于

示例：

```
SELECT inet '192.168.1.5' <> inet '192.168.1.4' AS RESULT;
 result
--------
 t
(1 row)
```

<<

描述：包含于

示例：

```
SELECT inet '192.168.1.5' << inet '192.168.1/24' AS RESULT;
 result
--------
 t
(1 row)
```

<<=

描述：包含于或等于

示例：

```
SELECT inet '192.168.1/24' <<= inet '192.168.1/24' AS RESULT;
 result
--------
 t
(1 row)
```

>>

描述：包含

示例：

```
SELECT inet '192.168.1/24' >> inet '192.168.1.5' AS RESULT;
 result
--------
 t
(1 row)
```

>>=

描述：包含或等于

示例：

```
SELECT inet '192.168.1/24' >>= inet '192.168.1/24' AS RESULT;
 result
--------
 t
(1 row)
```

~

描述：位非

示例：

```
SELECT ~ inet '192.168.1.6' AS RESULT;
    result
---------------
 63.87.254.249
(1 row)
```

&

描述：两个网络地址的每一位都进行“与”操作。

示例：

```
SELECT inet '192.168.1.6' & inet '10.0.0.0' AS RESULT;
 result
---------
 0.0.0.0
(1 row)
```

|

描述：两个网络地址的每一位都进行“或”操作。

示例：

```
SELECT inet '192.168.1.6' | inet '10.0.0.0' AS RESULT;
   result
-------------
 202.168.1.6
(1 row)
```

+

描述：加

示例：

```
SELECT inet '192.168.1.6' + 25 AS RESULT;
    result
```



```
--------------
 192.168.1.31
(1 row)
```

-

描述：减

示例：

```
SELECT inet '192.168.1.43' - 36 AS RESULT;
   result
-------------
 192.168.1.7
(1 row)
SELECT inet '192.168.1.43' - inet '192.168.1.19' AS RESULT;
 result
--------
     24
(1 row)
```

6.25.2 网络地址函数

函数abbrev，host，text主要是为了提供可选的显示格式。

任何cidr值都能以显式或者隐式的方式转换为inet值，因此能够操作inet值的函数也同样能够操作cidr值。inet值也可以转换为cidr值，此时inet子网掩码右侧的所有位都将转换为零，以创建一个有效的cidr值。另外，用户还可以使用常规的类型转换语法将一个文本字符串转换为inet或cidr值。例如：inet(expression)或colname::cidr。

abbrev(inet)

描述：缩写显示格式文本。

返回类型：text

示例：

```
SELECT abbrev(inet '10.1.0.0/16') AS RESULT;
   result
-------------
 10.1.0.0/16
(1 row)
```

abbrev(cidr)

描述：缩写显示格式文本。

返回类型：text

示例：

```
SELECT abbrev(cidr '10.1.0.0/16') AS RESULT;
 result
---------
 10.1/16
(1 row)
```

broadcast(inet)

描述：网络广播地址。

返回类型：inet

示例：

```
SELECT broadcast('192.168.1.5/24') AS RESULT;
      result
------------------
 192.168.1.255/24
(1 row)
```

family(inet)

描述：抽取地址族，4为IPv4，6为IPv6。

返回类型：int

示例：

```
SELECT family('::1') AS RESULT;
 result
--------
      6
(1 row)
```

host(inet)

描述：将主机地址类型抽出为文本。

返回类型：text

示例：

```
SELECT host('192.168.1.5/24') AS RESULT;
   result
-------------
 192.168.1.5
(1 row)
```

hostmask(inet)

描述：为网络构造主机掩码。

返回类型：inet

示例：

```
SELECT hostmask('192.168.23.20/30') AS RESULT;
 result
---------
 0.0.0.3
(1 row)
```

masklen(inet)

描述：抽取子网掩码长度。

返回类型：int

示例：

```
SELECT masklen('192.168.1.5/24') AS RESULT;
 result
--------
     24
(1 row)
```


netmask(inet)

描述：为网络构造子网掩码。

返回类型：inet

示例：

```
SELECT netmask('192.168.1.5/24') AS RESULT;
   result
---------------
 255.255.255.0
(1 row)
```

network(inet)

描述：抽取地址的网络部分。

返回类型：cidr

示例：

```
SELECT network('192.168.1.5/24') AS RESULT;
    result
----------------
 192.168.1.0/24
(1 row)
```

set_masklen(inet, int)

描述：为inet数值设置子网掩码长度。

返回类型：inet

示例：

```
SELECT set_masklen('192.168.1.5/24', 16) AS RESULT;
    result
----------------
 192.168.1.5/16
(1 row)
```

set_masklen(cidr, int)

描述：为cidr数值设置子网掩码长度。

返回类型：cidr

示例：

```
SELECT set_masklen('192.168.1.0/24'::cidr, 16) AS RESULT;
    result
----------------
 192.168.0.0/16
(1 row)
```

text(inet)

描述：把IP地址和掩码长度抽取为文本。

返回类型：text

示例：


```
SELECT text(inet '192.168.1.5') AS RESULT;
    result
----------------
 192.168.1.5/32
(1 row)
```

trunc(macaddr)

函数trunc(macaddr)返回一个MAC地址，该地址的最后三个字节设置为零。

描述：把后三个字节置为零。

返回类型：macaddr

示例：

```
SELECT trunc(macaddr '12:34:56:78:90:ab') AS RESULT;
     result
-------------------
 12:34:56:00:00:00
(1 row)
```

macaddr类型还支持标准关系操作符（>，<=等）用于词法排序，和按位运算符（~，&和|）非，与和或。

6.26 系统信息函数

6.26.1 会话信息函数

current_catalog

描述：当前数据库的名字（在标准SQL中称“catalog”），与current_database()同义。

返回值类型：name

示例：

```
SELECT current_catalog;
 current_database
------------------
 gaussdb
(1 row)
```

current_database()

描述：当前数据库的名字。

返回值类型：name

示例：

```
SELECT current_database();
 current_database
------------------
 gaussdb
(1 row)
```


current_query()

描述：由客户端提交的当前执行语句（可能包含多个声明）。

返回值类型：text

示例：

```
SELECT current_query();
     current_query
-------------------------
 SELECT current_query();
(1 row)
```

current_schema[()]

描述：当前模式的名字。current_schema返回在搜索路径中第一个顺位有效的模式名。（如果搜索路径为空则返回NULL，没有有效的模式名也返回NULL）。如果创建表或者其他命名对象时没有声明目标模式，则将使用这些对象的模式。

返回值类型：name

示例：

```
SELECT current_schema();
 current_schema
----------------
 public
(1 row)
```

current_schemas(boolean)

描述：current_schemas(boolean)返回搜索路径中所有模式名字的数组。布尔选项决定像pg_catalog这样隐含包含的系统模式是否包含在返回的搜索路径中。

返回值类型：name[]

示例：

```
SELECT current_schemas(true);
   current_schemas
---------------------
 {pg_catalog,public}
(1 row)
```

说明

搜索路径可以通过运行时设置更改。命令是：

```
SET search_path TO schema [, schema, ...]
```

current_user

描述：当前执行环境下的用户名。current_user是用于表示权限检查的用户标识。通常用来表示会话用户，但是可以通过**SET ROLE**改变。在函数执行的过程中随着属性SECURITY DEFINER的改变，其值也会改变。

返回值类型：name

示例：

```
SELECT current_user;
 current_user
```

```
--------------
 dbadmin
(1 row)
```

inet_client_addr()

描述：显示当前连接的客户端IP信息。

说明

- 此函数只有在远程连接模式下有效。
- 如果是通过本地连接，使用此接口显示为空。

返回值类型：inet

示例：

```
SELECT inet_client_addr();
 inet_client_addr
------------------
 10.10.0.50
(1 row)
```

inet_client_port()

描述：显示当前连接的客户端的端口号。

说明

此函数只有在远程连接模式下有效。

返回值类型：integer

示例：

```
SELECT inet_client_port();
 inet_client_port
------------------
            33143
(1 row)
```

inet_server_addr()

描述：显示当前服务器的IP信息。

说明

- 此函数只有在远程连接模式下有效。
- 如果是通过本地连接，使用此接口显示为空。

返回值类型：inet

示例：

```
SELECT inet_server_addr();
 inet_server_addr
------------------
 10.10.0.13
(1 row)
```


inet_server_port()

描述：显示当前服务器的端口。如果是通过Unix-domain socket连接的，则所有这些函数都返回NULL。

说明

此函数只有在远程连接模式下有效。

返回值类型：integer

示例：

```
SELECT inet_server_port();
 inet_server_port
------------------
 8000
(1 row)
```

pg_backend_pid()

描述：当前会话连接的服务进程的进程ID。

返回值类型：integer

示例：

```
SELECT pg_backend_pid();
 pg_backend_pid
-----------------
 140229352617744
(1 row)
```

pg_conf_load_time()

描述：配置加载时间。pg_conf_load_time返回最后加载服务器配置文件的时间戳。

返回值类型：timestamp with time zone

示例：

```
SELECT pg_conf_load_time();
      pg_conf_load_time
-------------------------------
 2017-09-01 16:05:23.89868+08
(1 row)
```

pg_my_temp_schema()

描述：pg_my_temp_schema返回当前会话中临时模式的OID，如果不存在（没有创建临时表）的话则返回0。如果给定的OID是其它会话中临时模式的OID，pg_is_other_temp_schema则返回true。

返回值类型：oid

示例：

```
SELECT pg_my_temp_schema();
 pg_my_temp_schema
-------------------
                 0
(1 row)
```


pg_is_other_temp_schema(oid)

描述：是否为另一个会话的临时模式。

返回值类型：boolean

示例：

```
SELECT pg_is_other_temp_schema(25356);
 pg_is_other_temp_schema
-------------------------
 f
(1 row)
```

pg_postmaster_start_time()

描述：服务器启动时间。pg_postmaster_start_time返回服务器启动时的timestamp with time zone。

返回值类型：timestamp with time zone

示例：

```
SELECT pg_postmaster_start_time();
   pg_postmaster_start_time
------------------------------
 2017-08-30 16:02:54.99854+08
(1 row)
```

pg_trigger_depth()

描述：触发器的嵌套层次。

返回值类型：integer

示例：

```
SELECT pg_trigger_depth();
 pg_trigger_depth
------------------
                0
(1 row)
```

pgxc_version()

描述：Postgres-XC版本信息。

返回值类型：text

示例：

```
SELECT pgxc_version();
                                                pgxc_version
-------------------------------------------------------------------------------------------------------------
 Postgres-XC 1.1 on x86_64-unknown-linux-gnu, based on PostgreSQL 9.2.4, compiled by g++ (GCC) 5.4.0, 64-bit
(1 row)
```

session_user

描述：查询会话用户名。session_user通常是连接当前数据库的初始用户，不过系统管理员可以用**SET SESSION AUTHORIZATION**修改这个设置。

返回值类型：name

示例：

```
SELECT session_user;
 session_user
--------------
 dbadmin
(1 row)
```

user

描述：等价于current_user。

返回值类型：name

示例：

```
SELECT user;
 current_user
--------------
 dbadmin
(1 row)
```

version()

描述：版本信息。version返回一个描述服务器版本信息的字符串。

返回值类型：text

示例：

```
SELECT version();
                                                         version
---------------------------------------------------------------------------------------------------------------------------------
 PostgreSQL 9.2.4 gsql ((GaussDB 8.1.3 build 39137c2d) compiled at 2022-04-01 15:43:11 commit 3629 last mr 5138 release) on x86_64-unknown-linux-gnu, compiled by g++ (GCC) 5.4.0, 64-bit
(1 row)
```

6.26.2 访问权限查询函数

has_any_column_privilege(user, table, privilege)

描述：指定用户是否有访问表任何列的权限。

参数：user可以通过名字（text类型）或OID来声明。table可以通过名字（text类型）或者OID来声明。privilege使用文本字符串（text类型）来声明，该文本字符串可取值SELECT、INSERT、UPDATE、REFERENCES，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

has_any_column_privilege(table, privilege)

描述：当前用户是否有访问表任何列的权限。

参数：table可以通过名字（text类型）或者OID来声明。privilege使用文本字符串（text类型）来声明，该文本字符串可取值SELECT、INSERT、UPDATE、REFERENCES，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

备注：has_any_column_privilege检查用户是否以特定方式访问表的任何列。其参数可能与has_table_privilege类似，除了访问权限类型必须是SELECT、INSERT、UPDATE或REFERENCES的一些组合。

说明

拥有表的表级别权限则隐含的拥有该表每列的列级权限，因此如果与has_table_privilege参数相同，has_any_column_privilege总是返回true。但是如果授予至少一列的列级权限也返回成功。

has_column_privilege(user, table, column, privilege)

描述：指定用户是否有访问列的权限。

参数：user可以通过名字（text类型）或OID来声明。table可以通过名字（text类型）或者OID来声明。column可以通过列名（text类型）或属性号（smallint类型）来声明。privilege使用文本字符串来声明，该文本字符串可取值SELECT、INSERT、UPDATE、REFERENCES，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

has_column_privilege(table, column, privilege)

描述：当前用户是否有访问列的权限。

参数：table可以通过名字（text类型）或者OID来声明。column可以通过列名（text类型）或属性号（smallint类型）来声明。privilege使用文本字符串来声明，该文本字符串可取值SELECT、INSERT、UPDATE、REFERENCES，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

备注：has_column_privilege检查用户是否以特定方式访问一列。其参数类似于has_table_privilege，可以通过列名或属性号添加列。想要的访问权限类型必须是SELECT、INSERT、UPDATE或REFERENCES的一些组合。

说明

拥有表的表级别权限则隐含的拥有该表每列的列级权限。

has_database_privilege(user, database, privilege)

描述：指定用户是否有访问数据库的权限。

参数：user可以通过名字（text类型）或OID来声明。database可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值SELECT、INSERT、UPDATE、REFERENCES，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

has_database_privilege(database, privilege)

描述：当前用户是否有访问数据库的权限。

参数：database可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值CREATE、CONNECT、TEMPORARY或TEMP，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

备注：has_database_privilege检查用户是否能以在特定方式访问数据库。其参数类似has_table_privilege。访问权限类型必须是CREATE、CONNECT、TEMPORARY或TEMP（等价于TEMPORARY）的一些组合。

has_foreign_data_wrapper_privilege(user, fdw, privilege)

描述：指定用户是否有访问外部数据封装器的权限。

参数：user可以通过名字（text类型）或OID来声明。fdw表示外部数据封装器，可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串必须是USAGE。

参数fdw，表示外部数据封装器，取值范围为外部数据封装器名字或ID。

返回类型：boolean

has_foreign_data_wrapper_privilege(fdw, privilege)

描述：当前用户是否有访问外部数据封装器的权限。

参数：fdw表示外部数据封装器，可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串必须是USAGE。

返回类型：boolean

备注：has_foreign_data_wrapper_privilege检查用户是否能以特定方式访问外部数据封装器。其参数类似has_table_privilege。访问权限类型必须是USAGE。

has_function_privilege(user, function, privilege)

描述：指定用户是否有访问函数的权限。

参数：user可以通过名字（text类型）或OID来声明。function可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串必须是EXECUTE。

返回类型：boolean

has_function_privilege(function, privilege)

描述：当前用户是否有访问函数的权限。

参数：function可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串必须是EXECUTE。

返回类型：boolean

备注：has_function_privilege检查当前用户是否能以指定方式访问指定函数。其参数类似has_table_privilege。使用文本字符而不是OID声明一个函数时，允许输入的类型和regprocedure数据类型一样（请参考**对象标识符类型**）。访问权限类型必须是EXECUTE。

has_language_privilege(user, language, privilege)

描述：指定用户是否有访问语言的权限。

参数：user可以通过名字（text类型）或OID来声明。language可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串必须是USAGE。

返回类型：boolean

has_language_privilege(language, privilege)

描述：当前用户是否有访问语言的权限。

参数：language可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串必须是USAGE。

返回类型：boolean

备注：has_language_privilege检查用户是否能以特定方式访问过程语言。其参数类似has_table_privilege。访问权限类型必须是USAGE。

has_schema_privilege(user, schema, privilege)

描述：指定用户是否有访问模式的权限。

参数：user可以通过名字（text类型）或OID来声明。schema可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值CREATE、USAGE，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

has_schema_privilege(schema, privilege)

描述：当前用户是否有访问模式的权限。

参数：schema可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值CREATE、USAGE，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

备注：has_schema_privilege检查用户是否能以特定方式访问指定模式。其参数类似has_table_privilege。访问权限类型必须是CREATE或USAGE的一些组合。

has_server_privilege(user, server, privilege)

描述：指定用户是否有访问外部服务的权限。

参数：user可以通过名字（text类型）或OID来声明。server可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串取值必须是USAGE。

返回类型：boolean

has_server_privilege(server, privilege)

描述：当前用户是否有访问外部服务的权限。

参数：server可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串取值必须是USAGE。

返回类型：boolean

备注：has_server_privilege检查用户是否能以指定方式访问指定外部服务器。其参数类似has_table_privilege。访问权限类型必须是USAGE。

has_table_privilege(user, table, privilege)

描述：指定用户是否有访问表的权限。

参数：user可以通过名字（text类型）或OID来声明。table可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES或TRIGGER，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

has_table_privilege(table, privilege)

描述：当前用户是否有访问表的权限。

参数：table可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES或TRIGGER，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

备注：has_table_privilege检查用户是否以特定方式访问表。用户可以通过名字或OID（pg_authid.oid）来指定，public表明PUBLIC伪角色，如果缺省该参数，则使用current_user。该表可以通过名字或者OID声明。如果用名字声明，则在必要时可以用模式进行修饰。如果使用文本字符串来声明权限类型，可以给权限类型添加WITH GRANT OPTION，用来测试权限是否拥有授权选项。当有多个权限类型时，拥有任何所列出的权限之一，则结果便为true。

示例：

```
SELECT has_table_privilege('tpcds.web_site', 'select');
 has_table_privilege
---------------------
 t
(1 row)

SELECT has_table_privilege('dbadmin', 'tpcds.web_site', 'select,INSERT WITH GRANT OPTION ');
 has_table_privilege
---------------------
 t
(1 row)
```

pg_has_role(user, role, privilege)

描述：指定用户是否有角色的权限。

参数：user可以通过名字（text类型）或OID来声明。role可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值MEMBER、USAGE，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

pg_has_role(role, privilege)

描述：当前用户是否有角色的权限。

参数：role可以通过名字（text类型）或者OID来声明。privilege使用文本字符串来声明，该文本字符串可取值MEMBER、USAGE，也可以用逗号分隔列出的多个权限类型。

返回类型：boolean

备注：pg_has_role检查用户是否能以特定方式访问角色。其参数类似has_table_privilege，除了public不能用做用户名。访问权限类型必须是MEMBER或USAGE的一些组合。 MEMBER表示的是角色中的直接或间接成员关系（也就是SET ROLE的权限），而USAGE表示无需通过SET ROLE也直接拥有角色的使用权限。

6.26.3 模式可见性查询函数

模式可见性查询函数

每个函数对数据库对象执行可见性检查。对于函数和操作符，如果在前面的搜索路径中没有相同的对象名称和参数的数据类型，则此对象是可见的。对于操作符类，则要同时考虑名字和相关索引的访问方法。

所有这些函数都需要使用OID来标识要需要检查的对象。如果用户想通过名字测试对象，则可使用OID别名类型（regclass、regtype、regprocedure、regoperator、regconfig或regdictionary）。

比如，如果一个表所在的模式在搜索路径中，并且在前面的搜索路径中没有同名的表，那么这个表是可见的。它等效于表可以不带明确模式修饰进行引用。比如，要列出所有可见表的名字：

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

pg_collation_is_visible(collation_oid)

描述：排序是否在搜索路径中可见。

返回类型：boolean

pg_conversion_is_visible(conversion_oid)

描述：转换是否在搜索路径中可见。

返回类型：boolean

pg_function_is_visible(function_oid)

描述：函数是否在搜索路径中可见。

返回类型：boolean

pg_opclass_is_visible(opclass_oid)

描述：操作符类是否在搜索路径中可见。

返回类型：boolean

pg_operator_is_visible(operator_oid)

描述：操作符是否在搜索路径中可见。

返回类型：boolean

pg_opfamily_is_visible(opclass_oid)

描述：操作符族是否在搜索路径中可见。

返回类型：boolean

pg_table_is_visible(table_oid)

描述：表是否在搜索路径中可见。

返回类型：boolean

pg_ts_config_is_visible(config_oid)

描述：该文本检索配置是否在搜索路径中可见。

返回类型：boolean

pg_ts_dict_is_visible(dict_oid)

描述：文本检索词典是否在搜索路径中可见。

返回类型：boolean

pg_ts_parser_is_visible(parser_oid)

描述：文本搜索解析是否在搜索路径中可见。

返回类型：boolean

pg_ts_template_is_visible(template_oid)

描述：文本检索模板是否在搜索路径中可见。

返回类型：boolean

pg_type_is_visible(type_oid)

描述：类型（或域）是否在搜索路径中可见。

返回类型：boolean

6.26.4 系统表信息函数

format_type(type_oid, typemod)

描述：获取数据类型的SQL名称。

返回类型：text

备注：

format_type通过数据类型的类型OID以及可能的类型修饰词，返回其SQL名称。如果不知道具体的修饰词，则在类型修饰词的位置传入NULL。类型修饰词一般只对有长度

限制的数据类型有意义。format_type所返回的SQL名称中包含数据类型的长度值，其大小是：实际存储长度len - sizeof(int32)，单位字节。数据存储时需要32位的空间来存储用户对数据类型的自定义长度信息，即实际存储长度要比用户定义长度多4个字节。在下例中，format_type返回的SQL名称为“character varying(6)”，6表示varchar类型的长度值是6字节，因此该类型的实际存储长度为10字节。

```
SELECT format_type((SELECT oid FROM pg_type WHERE typname='varchar'), 10);
    format_type
---------------------
 character varying(6)
(1 row)
```

pg_check_authid(role_oid)

描述：检查是否存在给定oid的角色名。

返回类型：bool

pg_describe_object(catalog_id, object_id, object_sub_id)

描述：获取数据库对象的描述。

返回类型：text

备注：pg_describe_object返回由目录OID，对象OID和一个（或许0个）子对象ID指定的数据库对象的描述。这有助于确认存储在pg_depend系统表中对象的身份。

pg_get_constraintdef(constraint_oid)

描述：获取约束的定义。

返回类型：text

pg_get_constraintdef(constraint_oid, pretty_bool)

描述：获取约束的定义。

返回类型：text

备注：pg_get_constraintdef和pg_get_indexdef分别从约束或索引上使用创建命令进行重构。

pg_get_expr(pg_node_tree, relation_oid)

描述：反编译表达式的内部形式，假设其中的任何Vars都引用第二个参数指定的关系。

返回类型：text

pg_get_expr(pg_node_tree, relation_oid, pretty_bool)

描述：反编译表达式的内部形式，假设其中的任何Vars都引用第二个参数指定的关系。

返回类型：text

备注：pg_get_expr反编译一个独立表达式的内部形式，比如一个字段的缺省值。在检查系统表的内容的时候很有用。如果表达式可能包含关键字，则指定它们引用相关的OID作为第二个参数；如果没有关键字，设置为零即可。

pg_get_indexdef(index_oid)

描述：获取索引的CREATE INDEX命令。

返回类型：text

index_oid为索引的OID，可以通过**PG_STATIO_ALL_INDEXES**系统视图查询。

示例：查询索引ds_ship_mode_t1_index1的OID及其创建命令。

```
SELECT indexrelid FROM PG_STATIO_ALL_INDEXES WHERE indexrelname = 'ds_ship_mode_t1_index1';
 indexrelid
------------
     136035
(1 row)
SELECT * FROM pg_get_indexdef(136035);
                                          pg_get_indexdef
-----------------------------------------------------------------------------------------------------------
 CREATE INDEX ds_ship_mode_t1_index1 ON tpcds.ship_mode_t1 USING psort (sm_ship_mode_sk)
TABLESPACE pg_default
(1 row)
```

pg_get_indexdef(index_oid, column_no, pretty_bool)

描述：获取索引的CREATE INDEX命令，或者如果column_no不为零，则只获取一个索引字段的定义。

返回类型：text

```
SELECT * FROM pg_get_indexdef(136035,0,false);
                                          pg_get_indexdef
-----------------------------------------------------------------------------------------------------------
 CREATE INDEX ds_ship_mode_t1_index1 ON tpcds.ship_mode_t1 USING psort (sm_ship_mode_sk)
TABLESPACE pg_default
(1 row)
SELECT * FROM pg_get_indexdef(136035,1,false);
 pg_get_indexdef
-----------------
 sm_ship_mode_sk
(1 row)
```

pg_get_keywords()

描述：获取SQL关键字和类别列表。

返回类型：setof record

备注：pg_get_keywords返回一组关于描述服务器识别SQL关键字的记录。word列包含关键字。catcode列包含一个分类代码：U表示通用的，C表示列名，T表示类型或函数名，或R表示保留。catdesc列包含了一个可能本地化描述分类的字符串。

pg_get_ruledef(rule_oid)

描述：获取规则的CREATE RULE命令。

返回类型：text

pg_get_ruledef(rule_oid, pretty_bool)

描述：获取规则的CREATE RULE命令。

返回类型：text

pg_get_userbyid(role_oid)

描述：获取给定OID的角色名。

返回类型：name

备注：pg_get_userbyid通过角色的OID抽取对应的用户名。

pg_get_viewdef(viewname text [, pretty bool [, fullflag bool]])

描述：为视图获取底层的SELECT命令。

返回类型：text

备注：

- pg_get_viewdef重构定义视图的SELECT查询。pretty bool参数为true时，显示格式“适合打印”，且该格式易读。pretty bool参数缺省值为false，显示格式不易读。如果用于转储，那么尽可能使用缺省格式。pretty bool参数只对有效视图生效。
- fullflag bool参数为true时，显示视图的完整定义。其缺省值为false。

pg_get_viewdef(viewoid oid [, pretty bool [, fullflag bool]])

描述：为视图获取底层的SELECT命令。

返回类型：text

pg_get_viewdef(view_oid, wrap_column_int)

描述：为视图获取底层的SELECT命令；行字段被换到指定的列数，打印是隐含的。

返回类型：text

pg_get_tabledef(table_oid)

描述：根据table_oid获取表定义。

返回类型：text

示例：先通过系统表pg_class获取表customer_t2的OID，再使用此函数查询表customer_t2的定义，可获取创建表customer_t2时的表字段，表的存储方式（行存或列存）及表的分布方式等信息。

```
SELECT oid FROM pg_class WHERE relname ='customer_t2';
  oid
-------
 17353
(1 row)

SELECT * FROM pg_get_tabledef(17353);
              pg_get_tabledef
----------------------------------------------
```



```
SET search_path = dbadmin;                +
CREATE  TABLE customer_t2 (               +
        state_id character(2),            +
        state_name character varying(40), +
        area_id numeric                   +
)                                         +
WITH (orientation=column, compression=low)+
DISTRIBUTE BY HASH(state_id)              +
TO GROUP group_version1;
(1 row)
```

pg_get_tabledef(table_name)

描述：根据table_name获取表定义。

返回类型：text

备注：pg_get_tabledef重构出表定义的CREATE语句，包含了表定义本身、索引信息、comments信息。对于表对象依赖的group、schema、tablespace、server等信息，需要用户自己去创建，表定义里不会有这些对象的创建语句。

pg_options_to_table(reloptions)

描述：获取存储选项名称/值对的集合。

返回类型：setof record

备注：pg_options_to_table当通过pg_class.reloptions或pg_attribute.attoptions时返回存储选项名字/值对（ option_name/option_value ）的集合。

示例：

```
CREATE TABLE customer_test
(
  state_ID   CHAR(2),
  state_NAME VARCHAR2(40),
  area_ID    NUMBER
)
WITH (ORIENTATION = COLUMN,COMPRESSION=middle);
SELECT pg_options_to_table(reloptions) FROM pg_class WHERE relname='customer_test';
pg_options_to_table
----------------------
 (orientation,column)
 (compression,middle)
 (bucketnums,16384)
 (colversion,2.0)
 (enable_delta,false)
(5 rows)
```

pg_typeof(any)

描述：获取任何值的数据类型。

返回类型：regtype

备注：

pg_typeof返回传递给他的值的数据类型OID。这可能有助于故障排除或动态构造SQL查询。声明此函数返回regtype，这是一个OID别名类型（请参考**对象标识符类型**）；这意味着它是一个为了比较而显示类型名字的OID。

示例：


```
SELECT pg_typeof(33);
 pg_typeof
-----------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

collation for (any)

描述：获取参数的排序。

返回类型：text

备注：

表达式collation for返回传递给他的值的排序。示例：

```
SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
------------------
 "default"
(1 row)
```

值可能是引号括起来的并且模式限制的。如果没有为参数表达式排序，则返回一个null值。如果参数不是排序的类型，则抛出一个错误。

getdistributekey(table_name)

描述：获取一个hash表的分布列。

返回类型：text

示例：

```
SELECT getdistributekey('item');
 getdistributekey
------------------
 i_item_sk
(1 row)
```

table_skewness(text)

描述：查看表数据在所有节点的占比。

参数：表示待查询表的表名，为text类型。

返回值类型：record

table_skewness(table_name text, column_name text[, row_num text])

描述：查看表里某列数据，按hash分布规则，在各节点的占比情况。结果以数据节点上的数据量排序。

参数：table_name为表名；column_name为列名；row_num表示查看当前列所有数据，可缺省，默认为0。非0时，表示抽取指定条数的数据查看占比情况（每次采样结果可能不相同，只保证采集row_num条数据进行占比计算）。

返回值类型：record

示例：

根据tx表中的a列，按hash分布，则在1，2，0编号的DN上分布的数据量分别为7条，2条，1条。

```
SELECT * FROM table_skewness('tx','a');
 seqnum | num |  ratio
--------+-----+----------
 1      | 7   | 70.000%
 2      | 2   | 20.000%
 0      | 1   | 10.000%
(3 row)
```

table_data_skewness(data_row record, locatorType "char")

描述：计算指定表中列拼接出的record，对应的桶分布索引。

参数：data_row表示指定表中列拼接出的record，locatorType表示分布规则，当前建议指定'H'，按hash分布计算。

返回值类型：smallint

示例：

计算tx表中a列拼接的record，按照hash分布规则对应的桶分布索引。

```
SELECT a, table_data_skewness(row(a), 'H') FROM tx;
 a | table_data_skewness
---+---------------------
 3 |                   0
 6 |                   2
 7 |                   2
 4 |                   1
 5 |                   1
(5 rows)
```

table_distribution(schemaname text, tablename text)

描述：查看指定表在各个节点上占用的存储空间。

参数：表示待查询表的模式名和表名，均为text类型。

返回值类型：record

说明

- 使用本函数查询指定表存储分布信息，需要具备指定表的SELECT权限。
- table_distribution性能比table_skewness更优，尤其是在大集群大数据量场景下，请优先考虑使用table_distribution函数。
- 当使用table_distribution并希望直观的看到空间占比时，可使用dnsize/(sum(dnsize) over ())的方式查看出具体的占比情况。

table_distribution(regclass)

描述：查看指定表在各个节点上占用的存储空间。

参数：表示待查询表的表名或OID，表名可以有模式名限定。为regclass类型。

返回值类型：record

说明

- 使用本函数查询指定表存储分布信息，需要具备指定表的SELECT权限。
- table_distribution性能比table_skewness更优，尤其是在大集群大数据量场景下，请优先考虑使用table_distribution函数。
- 当使用table_distribution并希望直观的看到空间占比时，可使用dnsize/(sum(dnsize) over ())的方式查看出具体的占比情况。

table_distribution()

描述：查看当前库中所有表在各节点的存储空间分布情况。

返回值类型：record

说明

- 使用本函数涉及全库表信息查询，需要具备管理员权限或预置角色gs_role_read_all_stats权限。

当前基于table_distribution()函数，GaussDB(DWS)提供视图PGXC_GET_TABLE_SKEWNESS进行数据倾斜查询，建议在数据库中表数量（小于10000）较少的场景直接使用。

示例：

```
SELECT * FROM table_distribution();
    schemaname     |        tablename        |   nodename   | dnsize
-------------------+-------------------------+--------------+--------
 scheduler         | pg_task                 | dn_6005_6006 |   8192
 public            | ocr_group               | dn_6005_6006 |   8192
 public            | ocr_item                | dn_6005_6006 |   8192
 sea               | ocr_group               | dn_6005_6006 |  16384
 sea               | ocr_item                | dn_6005_6006 |  16384
 public            | customer_t1             | dn_6005_6006 |  16384
 dbms_om           | gs_wlm_session_info     | dn_6005_6006 |   8192
 dbms_om           | gs_wlm_operator_info    | dn_6005_6006 |   8192
 dbms_om           | gs_wlm_ec_operator_info | dn_6005_6006 |   8192
 public            | pgxc_copy_error_log     | dn_6005_6006 |   8192
 information_schema | sql_features           | dn_6005_6006 |  98304
 information_schema | sql_implementation_info | dn_6005_6006 |  49152
 information_schema | sql_languages          | dn_6005_6006 |  49152
 information_schema | sql_packages           | dn_6005_6006 |  49152
 information_schema | sql_parts              | dn_6005_6006 |  49152
 information_schema | sql_sizing             | dn_6005_6006 |  49152
 information_schema | sql_sizing_profiles    | dn_6005_6006 |   8192
 scheduler         | pg_task                 | dn_6003_6004 |   8192
 public            | ocr_group               | dn_6003_6004 |   8192
 public            | ocr_item                | dn_6003_6004 |  16384
 sea               | ocr_group               | dn_6003_6004 |   8192
 sea               | ocr_item                | dn_6003_6004 |  16384
 public            | customer_t1             | dn_6003_6004 |  16384
 dbms_om           | gs_wlm_session_info     | dn_6003_6004 |   8192
 dbms_om           | gs_wlm_operator_info    | dn_6003_6004 |   8192
 dbms_om           | gs_wlm_ec_operator_info | dn_6003_6004 |   8192
 public            | pgxc_copy_error_log     | dn_6003_6004 |   8192
 information_schema | sql_features           | dn_6003_6004 |  98304
 information_schema | sql_implementation_info | dn_6003_6004 |  49152
 information_schema | sql_languages          | dn_6003_6004 |  49152
 information_schema | sql_packages           | dn_6003_6004 |  49152
 information_schema | sql_parts              | dn_6003_6004 |  49152
 information_schema | sql_sizing             | dn_6003_6004 |  49152
 information_schema | sql_sizing_profiles    | dn_6003_6004 |   8192
 scheduler         | pg_task                 | dn_6001_6002 |   8192
 public            | ocr_group               | dn_6001_6002 |  16384
 public            | ocr_item                | dn_6001_6002 |   8192
 sea               | ocr_group               | dn_6001_6002 |   8192
 sea               | ocr_item                | dn_6001_6002 |  16384
```

```
public            | customer_t1              | dn_6001_6002 |  16384
dbms_om           | gs_wlm_session_info      | dn_6001_6002 |   8192
dbms_om           | gs_wlm_operator_info     | dn_6001_6002 |   8192
dbms_om           | gs_wlm_ec_operator_info | dn_6001_6002 |   8192
public            | pgxc_copy_error_log      | dn_6001_6002 |   8192
information_schema | sql_features            | dn_6001_6002 |  98304
information_schema | sql_implementation_info | dn_6001_6002 |  49152
information_schema | sql_languages           | dn_6001_6002 |  49152
information_schema | sql_packages            | dn_6001_6002 |  49152
information_schema | sql_parts               | dn_6001_6002 |  49152
information_schema | sql_sizing              | dn_6001_6002 |  49152
information_schema | sql_sizing_profiles     | dn_6001_6002 |   8192
(51 rows)
```

gs_table_distribution(schemaname text, tablename text)

描述：快速查看指定表在各个节点上占用的存储空间。

返回值类型：record

表 6-19 gs_table_distribution(schemaname text, tablename text)

名称	类型	描述
schemaname	name	模式名称。
tablename	name	表名。
relkind	character	类型。 ● i: 索引 ● r: 表
nodename	name	节点名称。
dnsize	bigint	表在该节点上的存储空间大小，单位：字节。

说明

- 使用本函数查询指定表存储分布信息，需要具备指定表的SELECT权限。
- 该函数基于PG_RELFILENODE_SIZE系统表上的物理文件存储空间记录，需确保GUC参数use_workload_manager和enable_perm_space必须开启。
- 性能上，单表查询时，gs_table_distribution函数低于table_distribution函数；在全库表查询时，gs_table_distribution函数大幅度优于table_distribution函数；在大集群大数据量场景下，如果进行全库表表查询，建议优先使用gs_table_distribution函数。

gs_table_distribution()

描述：快速查看当前库中所有表在各节点的存储空间分布情况。

返回值类型：record

表 6-20 gs_table_distribution()

名称	类型	描述
schemaname	name	模式名称。
tablename	name	表名。
relkind	character	类型，i: 索引，r: 表。
nodename	name	节点名称。
dnsize	bigint	表在该节点上的存储空间大小，单位：字节。

说明

- 使用本函数查询指定表存储分布信息，需要具备指定表的SELECT权限。
- 该函数基于PG_RELFILENODE_SIZE系统表上的物理文件存储空间记录，需确保GUC参数use_workload_manager和enable_perm_space必须开启。
- 性能上，单表查询时，gs_table_distribution函数低于table_distribution函数；在全库表查询时，gs_table_distribution函数大幅度优于table_distribution函数；在大集群大数据量场景下，如果进行全库表表查询，建议优先使用gs_table_distribution函数。

pgxc_get_stat_dirty_tables(int dirty_percent, int n_tuples)

描述：获取各表的插入、更新、删除以及脏页率信息。该函数针对视图PGXC_GET_STAT_ALL_TABLES进行了性能优化，可以快速筛选出每个DN上满足脏页率大于dirty_percent，dead元组数大于n_tuples的表，将筛选结果返回到CN进行汇总并输出。

返回值类型：setof record

函数返回字段如下：

名称	类型	描述
relid	oid	表的OID
relname	name	表名
schemaname	name	表的模式名
n_tup_ins	bigint	插入的元组条数
n_tup_upd	bigint	更新的元组条数
n_tup_del	bigint	删除的元组条数
n_live_tup	bigint	live元组的条数
n_dead_tup	bigint	dead元组的条数
dirty_page_rate	numeric(5,2)	表的脏页率信息(%)

示例：

查询数据库内脏页率大于10%，脏数据行数大于1000行的表：

```
SELECT * FROM pgxc_get_stat_dirty_tables(0,0) where dirty_page_rate > 10 and n_dead_tup > 1000;
 relid |          relname          | schemaname | n_tup_ins | n_tup_upd | n_tup_del | n_live_tup | n_dead_tup |
dirty_page_rate
-------+---------------------------+------------+-----------+-----------+-----------+------------+------------
+-----------------
 16782 | bandwidth_history_table | scheduler  |    356355 |         0 |    202068 |     154287 |      29031 |
15.84
 12050 | gs_wlm_instance_history | pg_catalog |    406464 |         0 |    234835 |     171647 |      27721
|          13.90
(2 rows)
```

pgxc_get_stat_dirty_tables(int dirty_percent, int n_tuples, text schema)

描述：获取各表的插入、更新、删除以及脏页率信息。该函数针对视图PGXC_GET_STAT_ALL_TABLES进行了性能优化， 可以快速筛选出每个DN上满足脏页率大于dirty_percent，dead元组数大于n_tuples的表，将筛选结果返回到CN进行汇总并输出。模式名是schema的表。

返回值类型：setof record

函数返回字段同上述函数pgxc_get_stat_dirty_tables(int dirty_percent, int n_tuples)。

示例：

查询指定系统表pg_catalog.pg_class的脏页率：

```
SELECT relname AS table_name,dirty_page_rate FROM pgxc_get_stat_dirty_tables(0,0,'pg_catalog') WHERE
relname = 'pg_class';
 table_name | dirty_page_rate
------------+-----------------
 pg_class   |           16.46
(1 row)
```

gs_switch_relfilenode()

描述：交换两个表或分区的元信息（重分布工具内部使用，用户直接使用会有错误信息提示）。

返回值类型：integer

copy_error_log_create()

描述：创建COPY FROM容错机制所需要的错误表（public.pgxc_copy_error_log）。

返回值类型：boolean

说明

- 此函数会尝试创建public.pgxc_copy_error_log表，表的详细信息请参见表6-21。
- 在relname列上创建B-tree索引，并REVOKE ALL on public.pgxc_copy_error_log FROM public对错误表进行权限控制（与COPY语句权限一致）。
- 由于尝试创建的public.pgxc_copy_error_log定义是一张行存表，因此集群上必须支持行存表的创建才能够正常运行此函数，并使用后续的COPY容错功能。需要特别注意的是，enable_hadoop_env这个GUC参数开启后会禁止在集群内创建行存表（GaussDB(DWS)默认为off）。
- 此函数自身权限为Sysadmin及以上（与错误表、COPY权限一致）。
- 若创建前public.pgxc_copy_error_log表已存在或者copy_error_log_relname_idx索引已存在，则此函数会报错回滚。

表 6-21 错误表 public.pgxc_copy_error_log 信息

列名称	类型	描述
relname	varchar	表名称。以模式名.表名形式显示。
begintime	timestamp with time zone	出现数据格式错误的时间。
filename	character varying	出现数据格式错误的数据源文件名。
rownum	bigint	在数据源文件中，出现数据格式错误的行号。
rawrecord	text	在数据源文件中，出现数据格式错误的原始记录。为了防止字段长度过大，限制字段的长度不超过1024 byte。
detail	text	详细错误信息。

示例：

```
SELECT copy_error_log_create();
 copy_error_log_create
-----------------------
 t
(1 row)
```

6.26.5 系统函数信息函数

pv_builtin_functions()

描述：查询系统内置函数的信息。

返回类型：record

pg_get_functiondef(func_oid)

描述：获取函数的定义。

返回类型：text

func_oid为函数的OID，可以通过**PG_PROC**系统表查询。

示例：查询函数justify_days的OID及其函数定义。

```
SELECT oid FROM pg_proc WHERE proname ='justify_days';
 oid
------
 1295
(1 row)

SELECT * FROM pg_get_functiondef(1295);
 headerlines |                          definition
-------------+-------------------------------------------------------------
           4 | CREATE OR REPLACE FUNCTION pg_catalog.justify_days(interval)+
             |  RETURNS interval                                           +
             |  LANGUAGE internal                                            +
             |  IMMUTABLE STRICT NOT FENCED NOT SHIPPABLE                  +
             | AS $function$interval_justify_days$function$                +
             |
(1 row)
```

pg_get_function_arguments(func_oid)

描述：获取函数定义的参数列表（带默认值）。

返回类型：text

备注：pg_get_function_arguments返回一个函数的参数列表，需要在CREATE FUNCTION中使用这种格式。

pg_get_function_identity_arguments(func_oid)

描述：获取参数列表来确定一个函数 (不带默认值)。

返回类型：text

备注：pg_get_function_identity_arguments返回需要的参数列表用来标识函数，这种形式需要在ALTER FUNCTION中使用，并且这种形式省略了默认值。

pg_get_function_result(func_oid)

描述：获取函数的RETURNS子句。

返回类型：text

备注：pg_get_function_result为函数返回适当的RETURNS子句。

6.26.6 注释信息函数

col_description(table_oid, column_number)

描述：获取一个表字段的注释。

返回类型：text

备注：col_description返回一个表中字段的注释，通过表OID和字段号来声明。

示例：查询pg_class系统表获取表OID，查询INFORMATION_SCHEMA.COLUMNS系统视图获取column_number。

```
SELECT COLUMN_NAME,ORDINAL_POSITION FROM INFORMATION_SCHEMA.COLUMNS WHERE
TABLE_NAME = 't' AND COLUMN_NAME = 'a';
```



```
column_name | ordinal_position
-------------+------------------
a           |                1
(1 row)

SELECT oid FROM pg_class WHERE relname='t';
  oid
-------
 44020
(1 row)

SELECT col_description(44020,1);
    col_description
-----------------------
 This is a test table.
(1 row)
```

obj_description(object_oid, catalog_name)

描述：获取一个数据库对象的注释。

返回类型：text

备注：带有两个参数的obj_description返回一个数据库对象的注释，该对象是通过其OID和其所属的系统表名字声明。比如，obj_description(123456,'pg_class')将返回OID为123456的表的注释。只带一个参数的obj_description只要求对象OID。

obj_description不能用于表字段，因为字段没有自己的OID。

obj_description(object_oid)

描述：获取一个数据库对象的注释。

返回类型：text

shobj_description(object_oid, catalog_name)

描述：获取一个共享数据库对象的注释。

返回类型：text

备注：shobj_description和obj_description差不多，不同之处仅在于前者用于共享对象。一些系统表是通用于集群中所有数据库的全局表，因此这些表的注释也是全局存储的。

6.26.7 事务 ID 和快照

以下的函数在输出形式中提供服务器事务信息。这些函数的主要用途是为了确定在两个快照之间有哪个事务提交。

pgxc_is_committed(transaction_id)

描述：如果提交或忽略给定的XID（ gxid ）。NULL表示的状态是未知的（ 运行，准备，冻结等 ）。

返回类型：bool

txid_current()

描述：获取当前事务ID。

返回类型：bigint

txid_current_snapshot()

描述：获取当前快照。

返回类型：txid_snapshot

txid_snapshot_xip(txid_snapshot)

描述：在快照中获取正在进行的事务ID。

返回类型：setof bigint

txid_snapshot_xmax(txid_snapshot)

描述：获取快照的xmax。

返回类型：bigint

txid_snapshot_xmin(txid_snapshot)

描述：获取快照的xmin。

返回类型：bigint

txid_visible_in_snapshot(bigint, txid_snapshot)

描述：在快照中事务ID是否可见（不使用子事务ID）。

返回类型：boolean

内部事务ID类型（xid）是32位，每40亿事务一次循环。这些函数使用的数据类型txid_snapshot，存储在特定时刻事务ID可见性的信息。其组件描述在**表6-22**。

表 6-22 快照组件

名字	描述
xmin	最早的事务ID（txid）仍然活动。所有较早事务将是已经提交可见的，或者是直接回滚。
xmax	作为尚未分配的txid。所有大于或等于此txids的都是尚未开始的快照时间，因此不可见。
xip_list	当前快照中活动的txids。这个列表只包含在xmin和xmax之间活动的txids；有可能活动的txids高于xmax。 介于大于等于xmin、小于xmax，并且不在这个列表中的txid，在这个时间快照已经完成的，因此按照提交状态查看他是可见还是回滚。这个列表不包含子事务的txids。

txid_snapshot的文本表示为：xmin:xmax:xip_list。

示例：10:20:10,14,15意思为：xmin=10, xmax=20, xip_list=10, 14, 15。

6.26.8 计算子集群函数

pv_compute_pool_workload()

描述：返回计算子集群当前的负载状态。

返回类型：void

示例：

```
SELECT * from pv_compute_pool_workload();
 nodename  | rpinuse | maxrp | nodestate
-----------+---------+-------+-----------
 datanode1 |       0 |  1000 | normal
 datanode2 |       0 |  1000 | normal
(2 rows)
```

6.26.9 锁信息函数

pgxc_get_lock_conflicts()

描述：返回集群中有冲突的锁信息。当某个锁正在等待另一个锁，或正被另一个锁等待，则认为该锁是冲突的。

返回类型：setof record

6.27 系统管理函数

6.27.1 配置设置函数

配置设置函数是用于查询及修改运行时配置参数的函数。

current_setting(setting_name)

描述：当前的设置值。

返回值类型：text

备注：current_setting用于以查询形式获取setting_name的当前值。和SQL语句SHOW是等效的。比如：

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

set_config(setting_name, new_value, is_local)

描述：设置参数并返回新值。

返回值类型：text

备注：set_config将参数setting_name设置为new_value，如果is_local为true，则新值将只应用于当前事务。如果希望新值应用于当前会话，可以使用false，和SQL语句SET是等效的。比如：

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

6.27.2 通用文件访问函数

通用文件访问函数提供了对数据库服务器上的文件的本地访问接口。只有数据库集群目录和log_directory目录里面的文件可以访问。使用相对路径访问集群目录里面的文件，以及匹配log_directory配置而设置的路径访问日志文件。只有数据库系统管理员才能使用这些函数。

pg_ls_dir(dirname text)

描述：列出目录中的文件。

返回值类型：setof text

备注：pg_ls_dir返回指定目录里面的除了特殊项“.”和“..”之外所有名字。

示例：

```
SELECT pg_ls_dir('./');
      pg_ls_dir
----------------------
 .postgresql.conf.swp
 postgresql.conf
 pg_tblspc
 PG_VERSION
 pg_ident.conf
 core
 server.crt
 pg_serial
 pg_twophase
 postgresql.conf.lock
 pg_stat_tmp
 pg_notify
 pg_subtrans
 pg_ctl.lock
 pg_xlog
 pg_clog
 base
 pg_snapshots
 postmaster.opts
 postmaster.pid
 server.key.rand
 server.key.cipher
 pg_multixact
 pg_errorinfo
 server.key
 pg_hba.conf
 pg_replslot
 .pg_hba.conf.swp
 cacert.pem
 pg_hba.conf.lock
 global
 gaussdb.state
(32 rows)
```

pg_read_file(filename text, offset bigint, length bigint)

描述：返回一个文本文件的内容。

返回值类型：text

备注：pg_read_file返回一个文本文件的一部分，从offset开始，最多返回length字节（如果先达到文件结尾，则小于这个数值）。如果offset是负数，则它是相对于文件结尾回退的长度。如果省略了offset和length，则返回整个文件。

示例：

```
SELECT pg_read_file('postmaster.pid',0,100);
            pg_read_file
----------------------------------------
 53078                                 +
 /srv/BigData/hadoop/data1/coordinator+
 1500022474                            +
 8000                                  +
 /var/run/dws                          +
 localhost                             +
  2
(1 row)
```

pg_read_binary_file(filename text [, offset bigint, length bigint,missing_ok boolean])

描述：返回一个二进制文件的内容。

返回值类型：bytea

备注：pg_read_binary_file的功能与pg_read_file类似，除了结果的返回值为bytea类型不一致，相应地不会执行编码检查。与convert_from函数结合，这个函数可以用来读取用指定编码的一个文件。

```
SELECT convert_from(pg_read_binary_file('filename'), 'UTF8');
```

pg_stat_file(filename text)

描述：返回一个文本文件的状态信息。

返回值类型：record

备注：pg_stat_file返回一条记录，其中包含：文件大小、最后访问时间戳、最后更改时间戳、最后文件状态修改时间戳以及标识传入参数是否为目录的boolean值。典型的用法：

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

示例：

```
SELECT * FROM pg_stat_file('postmaster.pid');

 size |         access         |      modification      |         change
| creation | isdir
------+------------------------+------------------------+------------------------
+----------+-------
  117 | 2017-06-05 11:06:34+08 | 2017-06-01 17:18:08+08 | 2017-06-01 17:18:08+08
|          | f
(1 row)
SELECT (pg_stat_file('postmaster.pid')).modification;
      modification
------------------------
 2017-06-01 17:18:08+08
(1 row)
```


6.27.3 服务器信号函数

服务器信号函数向其他服务器进程发送控制信号。只有系统管理员才能使用这些函数。

pg_cancel_backend(pid int)

描述：取消一个后端的当前查询。

返回值类型：boolean

备注：pg_cancel_backend向由pid标识的后端进程发送一个查询取消（SIGINT）信号。一个活动的后端进程的PID可以从pg_stat_activity视图的pid字段找到，或者在服务器上用ps列出数据库进程。

示例：

```
SELECT pid FROM pg_stat_activity WHERE stmt_type ='RESET';
       pid
-----------------
 281471222065200
(1 row)

SELECT pg_cancel_backend(281471222065200);
 pg_cancel_backend
-------------------
 t
(1 row)
```

pg_reload_conf()

描述：服务器进程重新装载配置文件。

返回值类型：boolean

备注：pg_reload_conf给服务器发送一个SIGHUP信号，导致所有服务器进程重新装载配置文件。

示例：

```
SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

pg_rotate_logfile()

描述：滚动服务器的日志文件。

返回值类型：boolean

备注：pg_rotate_logfile通知日志文件管理器立即切换到一个新的输出文件。该函数仅在内置日志收集器运行时有效。

示例：

```
SELECT pg_rotate_logfile();
 pg_rotate_logfile
-------------------
 t
(1 row)
```


pg_terminate_backend(pid int)

描述：终止一个后台线程。

返回值类型：boolean

备注：如果成功，函数返回true，否则返回false。

示例：

```
SELECT pid FROM pg_stat_activity;
       pid
-----------------
 140657876268816
 140433774061312
 140433587902208
 140433656592128
 140433723717376
 140433637189376
 140433552770816
 140433481983744
 140433349310208
(9 rows)

SELECT pg_terminate_backend(140657876268816);
 pg_terminate_backend
----------------------
 t
(1 row)
```

pg_wlm_jump_queue(pid int)

描述：调整任务到CN队列的最前端。

返回值类型：boolean

备注：如果成功，函数返回true，否则返回false。

示例：

```
SELECT pid FROM pg_stat_activity WHERE stmt_type ='RESET';
       pid
-----------------
 281471222065200
(1 row)

SELECT pg_wlm_jump_queue(281471222065200);
 pg_wlm_jump_queue
-------------------
 t
(1 row)
```

gs_wlm_switch_cgroup(pid int, cgroup text)

描述：调整作业的优先级到新控制组。

返回值类型：boolean

备注：如果成功，函数返回true，否则返回false。

pg_cancel_query(queryId int)

描述：取消一个后端的当前查询。该函数8.1.2及以上版本支持。

返回值类型：boolean

备注：pg_cancel_query向由query_id标识的后端进程发送一个查询取消（SIGINT）信号。一个活动的后端进程的query_id可以从pg_stat_activity视图的query_id字段找到。

示例：

```
SELECT query_id FROM pgxc_stat_activity WHERE stmt_type ='RESET';
 query_id
----------
        0
        0
(2 rows)

SELECT pg_cancel_query(0);
 pg_cancel_query
-----------------
 f
(1 row)
```

pgxc_cancel_query(queryId int)

描述：取消当前集群下正在执行的查询。该函数8.1.2及以上版本支持。

返回值类型：boolean

备注：如果所有节点的查询均已取消，函数返回true，否则返回false。

示例：

```
SELECT query_id FROM pgxc_stat_activity WHERE stmt_type ='RESET';
 query_id
----------
        0
        0
(2 rows)

SELECT pgxc_cancel_query(0);
 pgxc_cancel_query
-------------------
 f
(1 row)
```

pg_terminate_query(queryId int)

描述：终止一个后端的当前查询。该函数8.1.2及以上版本支持。

返回值类型：boolean

示例：

```
SELECT query_id FROM pgxc_stat_activity WHERE stmt_type ='RESET';
 query_id
----------
        0
        0
(2 rows)

SELECT pg_terminate_query(0);
 pg_terminate_query
--------------------
 f
(1 row)
```

pgxc_terminate_query(queryId int)

描述：终止当前集群下正在执行的查询。该函数8.1.2及以上版本支持。

返回值类型：boolean

示例：

```
SELECT query_id FROM pgxc_stat_activity;
    query_id
-----------------
 72339069014638631
(1 rows)

SELECT pgxc_terminate_query(72339069014638631);
 pgxc_terminate_query
----------------------
 t
(1 row)
```

6.27.4 快照同步函数

快照同步函数是导出当前快照的标识符。

create_wdr_snapshot()

描述：创建性能数据快照。

返回值类型：text

说明

- 该函数只有数据库管理员SYSADMIN才可以执行，非管理员执行会提示无权限。
- 该函数只能在CN上执行，在DN上执行会返回：“WDR snapshot can only be created on coordinator.”。
- 执行该函数前需确认enable_wdr_snapshot参数处于开启状态。如果enable_wdr_snapshot为off，执行该函数会返回：“WDR snapshot request can't be executed, because GUC parameter 'enable_wdr_snapshot' is off.”。
- 如果执行该函数时，快照线程由于节点重启等原因尚未启动，会提示错误：“WDR snapshot request can not be accepted, please retry later.”。
- 如果执行该函数失败，会提示：“Cannot respond to WDR snapshot request.”。
- 如果执行成功，会返回：“WDR snapshot request has been submitted.”。该提示表明创建快照请求已发送至后台快照线程，但不代表创建快照成功。

kill_snapshot(scope cstring)

描述：中止后台快照线程。该函数向后台快照线程发送中止信号并等待线程结束。

输入参数scope：表示操作范围。该参数取值范围为local和global。

- local表示中止当前CN上的快照线程。
- global表示不仅会中止当前CN上的快照线程，还会向集群中所有其他CN发送中止快照线程的请求，即中止集群中所有CN上的快照线程。
- 如果输入其他值，则报错“Scope is invalid, use "local" or "global".”。
- 输入参数可为空，表示默认取值为local。

返回值类型：无

说明

- 该函数只有数据库管理员SYSADMIN才有权执行，非管理员执行会提示无权限。
- 该函数只能在CN上执行，在DN上执行会提示：“kill_snapshot can only be executed on coordinator.”；
- 执行该函数会向后台快照线程发送中止信号并等待其结束。如果100s内快照线程仍未中止则会报错：“Kill snapshot thread failed”；

pg_export_snapshot()

描述：保存当前的快照并返回它的标识符。

返回值类型：text

备注：函数pg_export_snapshot保存当前的快照并返回一个文本字符串标识此快照。这个字符串必须传递给想要导入快照的客户端。可用在set transaction snapshot snapshot_id时导入snapshot，但是应用的前提是该事务设置了REPEATABLE READ隔离级别。该函数的输出不可用做set transaction snapshot的输入。

示例：

```
SELECT pg_export_snapshot();
 pg_export_snapshot
--------------------
 00000000000A7128-1
(1 row)
```

6.27.5 咨询锁函数

咨询锁函数用于管理咨询锁（Advisory Lock），此类函数目前仅内部使用。

pg_advisory_lock(key bigint)

描述：获取会话级别的排它咨询锁。

返回值类型：void

备注：pg_advisory_lock锁定应用程序定义的资源，该资源可以用一个64位或两个不重叠的32位键值标识。如果已经有另外的会话锁定了该资源，则该函数将阻塞到该资源可用为止。这个锁是排它的。多个锁定请求将会被压入栈中，因此，如果同一个资源被锁定了三次，它必须被解锁三次以将资源释放给其他会话使用。

pg_advisory_lock(key1 int, key2 int)

描述：获取会话级别的排它咨询锁。

返回值类型：void

pg_advisory_lock_shared(key bigint)

描述：获取会话级别的共享咨询锁。

返回值类型：void

pg_advisory_lock_shared(key1 int, key2 int)

描述：获取会话级别的共享咨询锁。

返回值类型：void

备注：pg_advisory_lock_shared类似于pg_advisory_lock，不同之处仅在于共享锁会话可以和其他请求共享锁的会话共享资源，但排它锁除外。

pg_advisory_unlock(key bigint)

描述：释放会话级别的排它咨询锁。

返回值类型：boolean

pg_advisory_unlock(key1 int, key2 int)

描述：释放会话级别的排它咨询锁。

返回值类型：boolean

备注：pg_advisory_unlock释放先前取得的排它咨询锁。如果释放成功则返回true。如果实际上并未持有指定的锁，将返回false并在服务器中产生一条SQL警告信息。

pg_advisory_unlock_shared(key bigint)

描述：释放会话级别的共享咨询锁。

返回值类型：boolean

pg_advisory_unlock_shared(key1 int, key2 int)

描述：释放会话级别的共享咨询锁。

返回值类型：boolean

备注：pg_advisory_unlock_shared类似于pg_advisory_unlock，不同之处在于该函数释放的是共享咨询锁。

pg_advisory_unlock_all()

描述：释放当前会话持有的所有咨询锁。

返回值类型：void

备注：pg_advisory_unlock_all将会释放当前会话持有的所有咨询锁，该函数在会话结束的时候被隐含调用，即使客户端异常地断开连接也是一样。

pg_advisory_xact_lock(key bigint)

描述：获取事务级别的排它咨询锁。

返回值类型：void

pg_advisory_xact_lock(key1 int, key2 int)

描述：获取事务级别的排它咨询锁。

返回值类型：void

备注：pg_advisory_xact_lock类似于pg_advisory_lock，不同之处在于锁是自动在当前事务结束时释放，而且不能被显式的释放。

pg_advisory_xact_lock_shared(key bigint)

描述：获取事务级别的共享咨询锁。

返回值类型：void

pg_advisory_xact_lock_shared(key1 int, key2 int)

描述：获取事务级别的共享咨询锁。

返回值类型：void

备注：pg_advisory_xact_lock_shared类似于pg_advisory_lock_shared，不同之处在于锁是在当前事务结束时自动释放，而且不能被显式的释放。

pg_try_advisory_lock(key bigint)

描述：尝试获取会话级排它咨询锁。

返回值类型：boolean

备注：pg_try_advisory_lock类似于pg_advisory_lock，不同之处在于该函数不会阻塞以等待资源的释放。它要么立即获得锁并返回true，要么返回false表示目前不能锁定。

pg_try_advisory_lock(key1 int, key2 int)

描述：尝试获取会话级排它咨询锁。

返回值类型：boolean

pg_try_advisory_lock_shared(key bigint)

描述：尝试获取会话级共享咨询锁。

返回值类型：boolean

pg_try_advisory_lock_shared(key1 int, key2 int)

描述：尝试获取会话级共享咨询锁。

返回值类型：boolean

备注：pg_try_advisory_lock_shared类似于pg_try_advisory_lock，不同之处在于该函数尝试获得共享锁而不是排它锁。

pg_try_advisory_xact_lock(key bigint)

描述：尝试获取事务级别的排它咨询锁。

返回值类型：boolean

pg_try_advisory_xact_lock(key1 int, key2 int)

描述：尝试获取事务级别的排它咨询锁。

返回值类型：boolean

备注：pg_try_advisory_xact_lock类似于pg_try_advisory_lock，不同之处在于如果得到锁，在当前事务的结束时自动释放，而且不能被显式的释放。

pg_try_advisory_xact_lock_shared(key bigint)

描述：尝试获取事务级别的共享咨询锁。

返回值类型：boolean

pg_try_advisory_xact_lock_shared(key1 int, key2 int)

描述：尝试获取事务级别的共享咨询锁。

返回值类型：boolean

备注：pg_try_advisory_xact_lock_shared类似于pg_try_advisory_lock_shared，不同之处在于如果得到锁，在当前事务结束时自动释放，而且不能被显式的释放。

6.27.6 复制函数

复制函数是系统为实现高可用在各个实例间进行日志同步或数据同步所提供的统计或操作方法。

说明

除统计查询外的复制函数为内部调用函数，不建议用户直接使用。

pg_create_logical_replication_slot('slot_name', 'plugin_name')

描述：创建逻辑复制槽。

参数说明：

- slot_name
  流复制槽名称。
  取值范围：字符串，不支持除字母，数字，以及（ _?-. ）以外的字符。
- plugin_name
  插件名称。
  取值范围：字符串，当前只支持“mppdb_decoding”。

返回值类型：name, text

备注：第一个返回值表示slot_name，第二个返回值表示该逻辑复制槽解码的起始LSN位置。

pg_create_physical_replication_slot ('slot_name', isDummyStandby)

描述：创建物理复制槽。

参数说明：

- slot_name
  流复制槽名称。
  取值范围：字符串，不支持除字母，数字，以及（ _?-. ）以外的字符。

- isDummyStandby
  复制槽是否为从备。
  取值范围：布尔值，支持true或false。

返回值类型：name, text

备注：第一个返回值表示slot_name，第二个返回值表示该物理复制槽解码的起始LSN位置。

pg_get_replication_slots()

描述：显示当前DN上所有的复制槽信息。

返回值类型：record

函数返回信息如下：

表 6-23 pg_get_replication_slots()字段

名称	类型	描述
slot_name	text	复制槽的名称
plugin	name	逻辑复制槽对应的输出插件名
slot_type	text	复制槽的类型
datoid	oid	复制槽的数据库OID
active	boolean	复制槽是否为激活状态
xmin	xid	复制槽事务标识
catalog_xmin	text	逻辑复制槽对应的最早解码事务标识
restart_lsn	text	复制槽的Xlog文件信息
dummy_standby	boolean	复制槽是否为从备

示例：

```
SELECT * FROM pg_get_replication_slots();
    slot_name    | plugin | slot_type | datoid | active | xmin | catalog_xmin |    restart_lsn    | dummy_standby
-----------------+--------+-----------+--------+--------+------+--------------+-------------------+---------------
 gs_roach_common |        | physical  |      0 | f      |      |    602861775 | FFFFFFFF/FFFFFFFF | f
(1 row)
```

pg_drop_replication_slot('slot_name')

描述：删除流复制槽。

参数说明：

- slot_name
  流复制槽名称。
  取值范围：字符串，不支持除字母，数字，以及（ _?-. ）以外的字符。

返回值类型：void

pg_logical_slot_peek_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

描述：解码并不推进流复制槽（下次解码可以再次获取本次解出的数据）。

参数说明：

- slot_name

  流复制槽名称。

  取值范围：字符串，不支持除字母，数字，以及（_?-.）以外的字符。

- LSN

  日志的LSN，表示只解码小于等于此LSN的日志。

  取值范围：字符串（LSN，格式为xlogid/xrecoff），如'1/2AAFC60'。为NULL时表示不对解码截止的日志位置做限制。

- upto_nchanges

  解码条数（包含begin和commit）。假设一共有三条事务，分别包含3、5、7条记录，如果upto_nchanges为4，那么会解码出前两个事务共8条记录。解码完第二条事务时发现解码条数记录大于等于upto_nchanges，会停止解码。

  取值范围：非负整数。

  说明

  LSN和upto_nchanges中任一参数达到限制，解码都会结束。

- options：此项为可选参数。
  - include-xids

    解码出的data列是否包含xid信息。

    取值范围：0或1，默认值为1。

    - 0：设为0时，解码出的data列不包含xid信息。
    - 1：设为1时，解码出的data列包含xid信息。
  - skip-empty-xacts

    解码时是否忽略空事务信息。

    取值范围：0或1，默认值为0。

    - 0：设为0时，解码时不忽略空事务信息。
    - 1：设为1时，解码时会忽略空事务信息。
  - include-timestamp

    解码信息是否包含commit时间戳。

    取值范围：0或1，默认值为0。

    - 0：设为0时，解码信息不包含commit时间戳。
    - 1：设为1时，解码信息包含commit时间戳。

返回值类型：text, uint, text

备注：函数返回解码结果，每一条解码结果包含三列，对应上述返回值类型，分别表示LSN位置、xid和解码内容。

pg_logical_slot_get_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')

描述：解码并推进流复制槽。

参数说明： 与pg_logical_slot_peek_changes一致，详细内容请参见**pg_logical_slot_peek_changes('slot_name', 'LSN', upto_nchanges, 'options_name', 'options_value')**。

pg_replication_slot_advance ('slot_name', 'LSN')

描述：直接推进流复制槽到指定LSN，不输出解码结果。

参数说明：

- slot_name

  流复制槽名称。

  取值范围：字符串，不支持除字母，数字，以及（ _?-. ）以外的字符。

- LSN

  推进到的日志LSN位置，下次解码时只会输出提交位置比该LSN大的事务结果。如果输入的LSN比当前流复制槽记录的推进位置还要小，则直接返回；如果输入的LSN比当前最新物理日志LSN还要大，则推进到当前最新物理日志LSN。

  取值范围：字符串（ LSN，格式为xlogid/xrecoff ）。

返回值类型：name, text

备注：返回值分别对应slot_name和实际推进至的LSN。

pg_stat_get_data_senders()

描述：显示当前DN上所有的数据页复制发送线程的统计信息。

返回值类型：record

函数返回信息如下：

表 6-24 pg_stat_get_data_senders()字段

名称	类型	描述
pid	bigint	线程pid
sender_pid	integer	当前sender的pid
local_role	text	本地的角色
peer_role	text	对端的角色
state	text	当前sender的复制状态
catchup_start	timestamp with time zone	catchup启动的时间

名称	类型	描述
catchup_end	timestamp with time zone	catchup结束的时间
queue_size	text	数据队列大小
queue_lower_tail	text	数据队列尾1位置
queue_header	text	数据队列头位置
queue_upper_tail	text	数据队列尾2位置
send_position	text	发送端发送的位置
receive_position	text	接收端接收的位置
catchup_type	text	catchup方式为全量还是增量
catchup_bcm_filename	text	catchup当前执行的bcm文件
catchup_bcm_finished	integer	catchup已操作完成的bcm文件数量
catchup_bcm_total	integer	catchup总共需要操作的bcm文件数量
catchup_percent	text	catchup已经操作完成的百分比
catchup_remaining_time	text	catchup预估剩余时间

pg_stat_get_wal_senders()

描述：显示当前DN上所有的WAL复制发送线程的统计信息。

返回值类型：record

函数返回信息如下：

表 6-25 pg_stat_get_wal_senders()字段

名称	类型	描述
pid	bigint	线程pid
sender_pid	integer	当前sender的pid
local_role	text	本地的角色
peer_role	text	对端的角色
peer_state	text	对端的状态
state	text	当前sender的复制状态

名称	类型	描述
catchup_start	timestamp with time zone	catchup启动的时间
catchup_end	timestamp with time zone	catchup结束的时间
sender_sent_location	text	发送端发送的LSN位置
sender_write_location	text	发送端write的LSN位置
sender_flush_location	text	发送端flush的LSN位置
sender_replay_location	text	发送端replay的LSN位置
receiver_received_location	text	接收端received的LSN位置
receiver_write_location	text	接收端write的LSN位置
receiver_flush_location	text	接收端flush的LSN位置
receiver_replay_location	text	接收端replay的LSN位置
sync_percent	text	同步百分比
sync_state	text	同步状态（异步复制，同步复制，还是潜在同步者）
sync_priority	integer	同步复制的优先级（0表示异步）
sync_most_available	text	在备机同步失败时，是否阻塞主机
channel	text	WALSender的信道信息

pg_stat_get_wal_receiver()

描述：显示当前DN上所有的WAL复制接收线程的统计信息。

返回值类型：record

函数返回信息如下：

表 6-26 pg_stat_get_wal_receiver()

名称	类型	描述
receiver_pid	integer	当前receiver的pid
local_role	text	本地的角色
peer_role	text	远端的角色
peer_state	text	远端的状态
state	text	当前receiver的复制状态
sender_sent_location	text	发送端发送的LSN位置
sender_write_location	text	发送端write的LSN位置
sender_flush_location	text	发送端flush的LSN位置
sender_replay_location	text	发送端replay的LSN位置
receiver_received_location	text	接收端received的LSN位置
receiver_write_location	text	接收端write的LSN位置
receiver_flush_location	text	接收端flush的LSN位置
receiver_replay_location	text	接收端replay的LSN位置
sync_percent	text	同步百分比
channel	text	WALReceiver的信道信息

pg_stat_get_stream_replications()

描述：显示当前DN上所有的复制统计信息。

返回值类型：record

函数返回信息如下：

表 6-27 pg_stat_get_stream_replications()

名称	类型	描述
local_role	text	本地的角色
static_connections	integer	连接统计

名称	类型	描述
db_state	text	数据库状态
detail_information	text	详细信息

示例：

```
SELECT * FROM pg_stat_get_stream_replications();
 local_role | static_connections | db_state | detail_information
------------+--------------------+----------+--------------------
 Normal     |                  0 | Normal   | Normal
(1 row)
```

pg_stat_xlog_space()

描述：显示当前DN上Xlog空间使用信息。

返回值类型：record

函数返回信息如下：

表 6-28 pg_stat_xlog_space()

名称	类型	描述
xlog_files	bigint	pg_xlog目录下，去除backup、archive_status等子目录，所有识别为xlog文件的数目。
xlog_size	bigint	pg_xlog目录下，去除backup、archive_status等子目录，所有识别为xlog文件的文件大小之和，单位为MB。
other_size	bigint	pg_xlog目录下backup、archive_status等子目录文件的大小之和，单位为MB。

示例：

```
SELECT * FROM pg_stat_xlog_space();
 xlog_files | xlog_size | other_size
------------+-----------+------------
         79 |      1264 |          0
(1 row)
```

pgxc_stat_xlog_space()

描述：显示所有主DN上Xlog空间使用信息。

返回值类型：record

函数返回信息如下：

表 6-29 pgxc_stat_xlog_space()

名称	类型	描述
node_name	name	节点名称
xlog_files	bigint	pg_xlog目录下，去除backup、archive_status等子目录，所有识别为xlog文件的数目。
xlog_size	bigint	pg_xlog目录下，去除backup、archive_status等子目录，所有识别为xlog文件的文件大小之和，单位为MB。
other_size	bigint	pg_xlog目录下backup、archive_status等子目录文件的大小之和，单位为MB。

示例：

```
SELECT * FROM  pgxc_stat_xlog_space();
 node_name   | xlog_files | xlog_size | other_size
-------------+------------+-----------+------------
 dn_6001_6002 |         73 |      1168 |          0
 dn_6003_6004 |         73 |      1168 |          0
 dn_6005_6006 |         73 |      1168 |          0
 cn_5003      |         79 |      1264 |          0
 cn_5001      |         72 |      1152 |          0
 cn_5002      |         73 |      1168 |          0
(6 rows)
```

6.27.7 资源管理函数

资源管理模块相关函数介绍。

gs_wlm_readjust_user_space(oid)

描述：用户永久存储空间校准函数。入参为用户oid，入参为0的情况下，会校准所有用户的永久存储空间。

返回值类型：text

示例：

```
SELECT gs_wlm_readjust_user_space(0);
gs_wlm_readjust_user_space
----------------------------
Exec Success
(1 row)
```

pgxc_wlm_readjust_schema_space()

描述：Schema永久存储空间校准函数。

返回值类型：text

示例：

```
SELECT pgxc_wlm_readjust_schema_space();
pgxc_wlm_readjust_schema_space
```

```
--------------------------------
Exec Success
(1 row)
```

pgxc_wlm_readjust_relfilenode_size_table()

描述：空间统计校准函数，不重建PG_RELFILENODE_SIZE系统表，重新校准用户和schema空间。

说明

由于校准函数执行时与其余业务之间的事务隔离性，校准函数对其它的正在执行的业务不可见，导致校准函数会漏掉此类业务的空间变化，为避免校准之后出现空间差误，建议用户使用空间校准函数时，应选择空间大小稳定无变化的时候执行校准。

返回值类型：text

示例：

```
SELECT pgxc_wlm_readjust_schema_space();
pgxc_wlm_readjust_relfilenode_size_table
-----------------------------------------
Exec Success
(1 row)
```

pgxc_wlm_readjust_relfilenode_size_table(integer)

描述：空间统计校准函数。

说明

由于校准函数执行时与其余业务之间的事务隔离性，校准函数对其它的正在执行的业务不可见，导致校准函数会漏掉此类业务的空间变化，为避免校准之后出现空间差误，建议用户使用空间校准函数时，应选择空间大小稳定无变化的时候执行校准。

参数：取值范围为0~4，不同的入参值代表不同的校准粒度。

- 入参为0(默认入参为0)：不重建PG_RELFILENODE_SIZE系统表，重新校准用户和schema空间。
- 入参为1：重建PG_RELFILENODE_SIZE系统表，并且重新校准用户和schema空间。
- 入参为2：重建PG_RELFILENODE_SIZE系统表。
- 入参为3：重新校准schema空间。
- 入参为4：重新校准用户空间。

返回值类型：text

示例：

```
SELECT * FROM  pgxc_wlm_readjust_relfilenode_size_table(1);
   result
--------------
 Exec success
(1 row)
```

pgxc_wlm_get_schema_space(cstring)

描述：在CN上查询某个逻辑集群下各实例的Schema空间信息，入参为逻辑集群名称。

返回值类型：record

函数返回字段如下：

名称	类型	描述
schemaname	text	模式名
schemaid	oid	模式OID
databasename	text	数据库名
databaseid	oid	数据库OID
nodename	text	实例名称
nodegroup	text	节点组名称
usedspace	bigint	已使用的空间大小
permspace	bigint	空间上限值

示例：

```
SELECT * FROM pgxc_wlm_get_schema_space('group1');
    schemaname     | schemaid | databasename | databaseid |   nodename   |  nodegroup   | usedspace |
permspace
--------------------+----------+--------------+------------+--------------+--------------+-----------+-----------
 pg_catalog         |       11 | test1        |      16384 | datanode1    | installation |   9469952 |        -1
 public             |     2200 | postgres     |      15253 | datanode1    | installation |  25280512 |        -1
 pg_toast           |       99 | test1        |      16384 | datanode1    | installation |   1859584 |        -1
 cstore             |      100 | test1        |      16384 | datanode1    | installation |         0 |        -1
 data_redis         |    18106 | postgres     |      15253 | datanode1    | installation |    655360 |        -1
 data_redis         |    18116 | test1        |      16384 | datanode1    | installation |         0 |        -1
 public             |     2200 | test1        |      16384 | datanode1    | installation |     16384 |        -1
 dbms_om            |     3987 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 dbms_job           |     3988 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 dbms_om            |     3987 | test1        |      16384 | datanode1    | installation |         0 |        -1
 dbms_job           |     3988 | test1        |      16384 | datanode1    | installation |         0 |        -1
 sys                |    11693 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 sys                |    11693 | test1        |      16384 | datanode1    | installation |         0 |        -1
 utl_file           |    14644 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 utl_raw            |    14669 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 dbms_sql           |    14674 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 dbms_output        |    14662 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 dbms_random        |    14666 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 dbms_lob           |    14701 | postgres     |      15253 | datanode1    | installation |         0 |        -1
 information_schema |    14300 | postgres     |      15253 | datanode1    | installation |    294912 |        -1
 information_schema |    14300 | test1        |      16384 | datanode1    | installation |    294912 |        -1
 utl_file           |    14644 | test1        |      16384 | datanode1    | installation |         0 |        -1
 dbms_output        |    14662 | test1        |      16384 | datanode1    | installation |         0 |        -1
 dbms_random        |    14666 | test1        |      16384 | datanode1    | installation |         0 |        -1
 utl_raw            |    14669 | test1        |      16384 | datanode1    | installation |         0 |        -1
 dbms_sql           |    14674 | test1        |      16384 | datanode1    | installation |         0 |        -1
 dbms_lob           |    14701 | test1        |      16384 | datanode1    | installation |         0 |        -1
 pg_catalog         |       11 | postgres     |      15253 | datanode1    | installation |  13049856 |        -1
 redisuser          |    16387 | postgres     |      15253 | datanode1    | installation |    630784 |        -1
 pg_toast           |       99 | postgres     |      15253 | datanode1    | installation |   3080192 |        -1
 cstore             |      100 | postgres     |      15253 | datanode1    | installation |   2408448 |        -1
 pg_catalog         |       11 | test1        |      16384 | datanode2    | installation |   9469952 |        -1
 public             |     2200 | postgres     |      15253 | datanode2    | installation |  25214976 |        -1
 pg_toast           |       99 | test1        |      16384 | datanode2    | installation |   1859584 |        -1
 cstore             |      100 | test1        |      16384 | datanode2    | installation |         0 |        -1
 data_redis         |    18106 | postgres     |      15253 | datanode2    | installation |    655360 |        -1
```

```
data_redis          |    18116 | test1        |      16384 | datanode2    | installation |         0 |       -1
public              |     2200 | test1        |      16384 | datanode2    | installation |     16384 |       -1
dbms_om             |     3987 | postgres     |      15253 | datanode2    | installation |         0 |       -1
dbms_job            |     3988 | postgres     |      15253 | datanode2    | installation |         0 |       -1
dbms_om             |     3987 | test1        |      16384 | datanode2    | installation |         0 |       -1
dbms_job            |     3988 | test1        |      16384 | datanode2    | installation |         0 |       -1
```

pgxc_wlm_analyze_schema_space(cstring)

描述：在CN上查询某个逻辑集群下集群整体的Schema空间信息，入参为逻辑集群名称。

返回值类型：record

函数返回字段如下：

名称	类型	描述
schemaname	text	模式名
databasename	text	数据库名
nodegroup	text	节点组名称
total_value	bigint	该模式的集群空间总值
avg_value	bigint	该模式的各实例空间平均值
skew_percent	integer	倾斜率
extend_info	text	提供信息包括：单实例空间最大值、单实例空间最小值以及最大最小空间所在的实例名

示例：

```
SELECT * FROM pgxc_wlm_analyze_schema_space('group1');
    schemaname     | databasename |  nodegroup   | total_value | avg_value | skew_percent |
extend_info
--------------------+--------------+--------------+-------------+-----------+--------------
+-----------------------------------------------
 pg_catalog         | test1        | installation |    56819712 |   9469952 |            0 | min:9469952
datanode1,max:9469952 datanode1
 public             | postgres     | installation |   150495232 |  25082538 |            0 | min:24903680
datanode6,max:25280512 datanode1
 pg_toast           | test1        | installation |    11157504 |   1859584 |            0 | min:1859584
datanode1,max:1859584 datanode1
 cstore             | test1        | installation |           0 |         0 |            0 | min:0 datanode1,max:0 datanode1
 data_redis         | postgres     | installation |     1966080 |    327680 |           50 | min:0
datanode4,max:655360 datanode1
 data_redis         | test1        | installation |           0 |         0 |            0 | min:0 datanode1,max:0 datanode1
 public             | test1        | installation |       98304 |     16384 |            0 | min:16384
datanode1,max:16384 datanode1
 dbms_om            | postgres     | installation |           0 |         0 |            0 | min:0 datanode1,max:0
datanode1
 dbms_job           | postgres     | installation |           0 |         0 |            0 | min:0 datanode1,max:0
datanode1
 dbms_om            | test1        | installation |           0 |         0 |            0 | min:0 datanode1,max:0 datanode1
 dbms_job           | test1        | installation |           0 |         0 |            0 | min:0 datanode1,max:0 datanode1
 sys                | postgres     | installation |           0 |         0 |            0 | min:0 datanode1,max:0 datanode1
 sys                | test1        | installation |           0 |         0 |            0 | min:0 datanode1,max:0 datanode1
 utl_file           | postgres     | installation |           0 |         0 |            0 | min:0 datanode1,max:0 datanode1
```

```
 utl_raw             | postgres     | installation |           0 |         0 |           0 | min:0 datanode1,max:0 datanode1
 dbms_sql            | postgres     | installation |           0 |         0 |           0 | min:0 datanode1,max:0 datanode1
 dbms_output         | postgres     | installation |           0 |         0 |           0 | min:0 datanode1,max:0
datanode1
 dbms_random         | postgres     | installation |           0 |         0 |           0 | min:0 datanode1,max:0
datanode1
 dbms_lob            | postgres     | installation |           0 |         0 |           0 | min:0 datanode1,max:0
datanode1
 information_schema | postgres     | installation |     1769472 |    294912 |           0 | min:294912
datanode1,max:294912 datanode1
 information_schema | test1        | installation |     1769472 |    294912 |           0 | min:294912
datanode1,max:294912 datanode1
 utl_file            | test1        | installation |           0 |         0 |           0 | min:0 datanode1,max:0 datanode1
 dbms_output         | test1        | installation |           0 |         0 |           0 | min:0 datanode1,max:0
datanode1
 dbms_random         | test1        | installation |           0 |         0 |           0 | min:0 datanode1,max:0
datanode1
 utl_raw             | test1        | installation |           0 |         0 |           0 | min:0 datanode1,max:0 datanode1
 dbms_sql            | test1        | installation |           0 |         0 |           0 | min:0 datanode1,max:0 datanode1
 dbms_lob            | test1        | installation |           0 |         0 |           0 | min:0 datanode1,max:0 datanode1
 pg_catalog          | postgres     | installation |    75431936 |  12571989 |           3 | min:12124160
datanode4,max:13049856 datanode1
 redisuser           | postgres     | installation |     1884160 |    314026 |          50 | min:16384
datanode4,max:630784 datanode1
 pg_toast            | postgres     | installation |    17154048 |   2859008 |           7 | min:2637824
datanode4,max:3080192 datanode1
 cstore              | postgres     | installation |    15294464 |   2549077 |           5 | min:2408448
datanode1,max:2703360 datanode6
(31 rows)
```

gs_wlm_set_queryband_action(cstring,cstring,int4)

描述：设置query_band关联行为和次序。

返回值类型：boolean

函数入参字段如下：

名称	类型	描述
qband	cstring	query_band键值对，长度上限为63个字符。
action	cstring	query_band关联行为。
order	int4	query_band搜索次序，缺省参数，默认为-1。

示例：

```
SELECT * FROM gs_wlm_set_queryband_action('a=1','respool=p1');
 gs_wlm_set_queryband_action
-----------------------------
 t
(1 row)
SELECT * FROM gs_wlm_set_queryband_action('a=3','respool=p1;priority=rush',1);
 gs_wlm_set_queryband_action
-----------------------------
 t
(1 row)
```

gs_wlm_set_queryband_order(cstring,int4)

描述：设置query_band次序。

返回值类型：boolean

函数入参字段如下：

名称	类型	描述
qband	cstring	query_band键值对
order	int4	query_band搜索次序，缺省参数，默认为-1

示例：

```
SELECT * FROM gs_wlm_set_queryband_order('a=1',2);
 gs_wlm_set_queryband_action
-----------------------------
 t
(1 row)
```

gs_wlm_get_queryband_action(cstring)

描述：查询query_band关联行为和次序。

返回值类型：record

函数返回字段如下：

名称	类型	描述
qband	cstring	query_band键值对
respool_id	Oid	query band关联资源池OID
respool	text	query band关联资源池名
priority	text	query band关联队队列内优先级
qborder	int4	query_band搜索次序

示例：

```
SELECT * FROM gs_wlm_get_queryband_action('a=1');
qband | respool_id | respool | priority | qborder
-------+------------+---------+----------+---------
 a=1   |      16388 | p1      | Medium   |      -1
(1 row)
```

gs_cgroup_reload_conf()

描述：在当前实例上进行cgroup配置文件在线加载。

返回值类型：record

函数返回字段如下：

名称	类型	描述
node_name	text	实例名称
node_host	text	实例所在节点的IP地址
result	text	cgroup在线加载是否成功

示例：

```
SELECT * FROM gs_cgroup_reload_conf();
 node_name |    node_host     | result
-----------+------------------+---------
 cn_5001   | 192.168.178.35 | success
```

pgxc_cgroup_reload_conf()

描述：在系统所有实例上进行cgroup配置文件在线加载。

返回值类型：record

函数返回字段如下：

名称	类型	描述
node_name	text	实例名称
node_host	text	实例所在节点的IP地址
result	text	cgroup在线加载是否成功

示例：

```
SELECT * FROM pgxc_cgroup_reload_conf();
  node_name   |     node_host     | result
--------------+-------------------+---------
 dn_6025_6026 | 192.168.178.177 | success
 dn_6049_6050 | 192.168.179.79  | success
 dn_6051_6052 | 192.168.179.79  | success
 dn_6055_6056 | 192.168.179.79  | success
 dn_6067_6068 | 192.168.181.57  | success
 dn_6023_6024 | 192.168.178.39  | success
 dn_6009_6010 | 192.168.181.21  | success
 dn_6011_6012 | 192.168.181.21  | success
 dn_6015_6016 | 192.168.181.21  | success
 dn_6029_6030 | 192.168.178.177 | success
 dn_6031_6032 | 192.168.178.177 | success
 dn_6045_6046 | 192.168.179.45  | success
 cn_5001      | 192.168.178.35  | success
 cn_5003      | 192.168.178.39  | success
 dn_6061_6062 | 192.168.181.179 | success
 cn_5006      | 192.168.179.45  | success
 cn_5004      | 192.168.178.177 | success
 cn_5002      | 192.168.181.21  | success
 cn_5005      | 192.168.178.187 | success
 dn_6019_6020 | 192.168.178.39  | success
 dn_6007_6008 | 192.168.178.35  | success
 dn_6071_6072 | 192.168.181.57  | success
 dn_6003_6004 | 192.168.178.35  | success
```

```
 dn_6013_6014 | 192.168.181.21  | success
 dn_6035_6036 | 192.168.178.187 | success
 dn_6037_6038 | 192.168.178.187 | success
 dn_6001_6002 | 192.168.178.35  | success
 dn_6063_6064 | 192.168.181.179 | success
 dn_6005_6006 | 192.168.178.35  | success
 dn_6057_6058 | 192.168.181.179 | success
 dn_6069_6070 | 192.168.181.57  | success
 dn_6027_6028 | 192.168.178.177 | success
 dn_6059_6060 | 192.168.181.179 | success
 dn_6041_6042 | 192.168.179.45  | success
 dn_6043_6044 | 192.168.179.45  | success
 dn_6047_6048 | 192.168.179.45  | success
 dn_6033_6034 | 192.168.178.187 | success
 dn_6065_6066 | 192.168.181.57  | success
 dn_6021_6022 | 192.168.178.39  | success
 dn_6017_6018 | 192.168.178.39  | success
 dn_6039_6040 | 192.168.178.187 | success
 dn_6053_6054 | 192.168.179.79  | success
(42 rows)
```

pgxc_cgroup_reload_conf(text)

描述：在某个节点上进行cgroup配置文件在线加载。入参为节点的IP地址。

返回值类型：record

函数返回字段如下：

名称	类型	描述
node_name	text	实例名称
node_host	text	实例所在节点的IP地址
result	text	cgroup在线加载是否成功

示例：

```
SELECT * FROM pgxc_cgroup_reload_conf('192.168.178.35');
  node_name   |   node_host    | result
--------------+----------------+---------
 cn_5001      | 192.168.178.35 | success
 dn_6007_6008 | 192.168.178.35 | success
 dn_6003_6004 | 192.168.178.35 | success
 dn_6001_6002 | 192.168.178.35 | success
 dn_6005_6006 | 192.168.178.35 | success
(5 rows)
```

gs_wlm_node_recover(boolean isForce)

描述：动态资源管理模式下，对CCN管控计数和作业信息进行更新恢复。该函数仅支持管理员执行，通常用于CN实例故障重启后的实例恢复，由集群管理组件（CM）调用，不建议用户直接调用。具体功能如下：

- CN执行：通知CCN清理该CN执行的作业信息和作业对应的管控计数信息。
- CCN执行：重置管控计数信息，并从CN上获取最新的慢车道作业信息。

返回值类型：bool

gs_wlm_node_clean(cstring nodename)

描述：动态资源管理模式下，对CCN上入参指定的CN执行的作业信息和对应的管控计数信息进行清理。该函数仅支持管理员执行，通常用于CN实例故障重启后的实例恢复，由集群管理组件（CM）调用，正常情况下不建议用户直接调用。

返回值类型：bool

pg_stat_get_wlm_node_resource_info(int4)

描述：显示所有DN资源的汇总信息。

返回值类型：record

函数返回字段如下：

名称	类型	描述
min_mem_util	integer	DN最小内存使用率。
max_mem_util	integer	DN最大内存使用率。
min_cpu_util	integer	DN最小CPU使用率。
max_cpu_util	integer	DN最大CPU使用率。
min_io_util	integer	DN最小IO使用率。
max_io_util	integer	DN最大IO使用率。
phy_usemem_rate	integer	物理节点最大内存使用率。

pg_stat_get_workload_struct_info()

描述：定位CCN排队问题的负载管理函数。此函数为内部函数，如需使用请联系技术支持工程师。

返回值类型：record

6.27.8 其它函数

pgxc_pool_check()

描述：检查连接池中缓存的连接数据是否与pgxc_node一致。

返回值类型：boolean

示例：

```
SELECT pgxc_pool_check();
 pgxc_pool_check
-----------------
 t
(1 row)
```


pgxc_pool_reload()

描述：更新连接池中缓存的连接信息。

返回值类型：boolean

示例：

```
SELECT pgxc_pool_reload();
 pgxc_pool_reload
------------------
 t
(1 row)
```

pg_pool_validate(clear boolean, co_node_name cstring)

描述：清理CN上无效的后台线程（这些后台线程持有无效的pooler连接，这里无效的pooler连接指的是连接到当前DN备实例的连接）。

返回值类型：record

pg_nodes_memory()

描述：查看所有节点的内存占用。

返回值类型：record

示例：

```
SELECT * FROM pg_nodes_memory();
     node_name       | used_memory |        shared_buffer_cache         |      top_context_memory
---------------------+-------------+------------------------------------+------------------------------
 dn_6003_6004        | 353 MB      | 108 MB(Utilization: 512 MB/21.00%) | PgStat BackendStatus(101 MB)
                     |             |                                    | TopMemoryContext(59 MB)
                     |             |                                    | gs_signal(56 MB)
 dn_6005_6006        | 353 MB      | 202 MB(Utilization: 512 MB/39.00%) | PgStat BackendStatus(101 MB)
                     |             |                                    | TopMemoryContext(59 MB)
                     |             |                                    | gs_signal(56 MB)
 dn_6001_6002        | 351 MB      | 201 MB(Utilization: 512 MB/39.00%) | PgStat BackendStatus(101 MB)
                     |             |                                    | TopMemoryContext(58 MB)
                     |             |                                    | gs_signal(56 MB)
 cn_5001             | 79 MB       | 48 MB(Utilization: 256 MB/19.00%)  | CacheMemoryContext(22 MB)
                     |             |                                    | PgStat BackendStatus(19 MB)
                     |             |                                    | gs_signal(16 MB)
 cn_5002             | 77 MB       | 95 MB(Utilization: 256 MB/37.00%)  | CacheMemoryContext(21 MB)
                     |             |                                    | PgStat BackendStatus(19 MB)
                     |             |                                    | gs_signal(16 MB)
 cn_5003             | 67 MB       | 50 MB(Utilization: 256 MB/19.00%)  | CacheMemoryContext(26 MB)
                     |             |                                    | gs_signal(16 MB)
                     |             |                                    | TopMemoryContext(9732 KB)
(18 rows)
```

plan_seed()

描述：获取前一次查询语句的seed值（内部使用）。

返回值类型：integer

示例：

```
SELECT plan_seed();
 plan_seed
-----------
         0
(1 row)
```

pg_stat_get_env()

描述：提供获取当前节点的环境变量信息。

返回值类型：record

示例：

```
SELECT * FROM  pg_stat_get_env();
 node_name |   host    | process | port |   installpath    |       datapath        |            log_directory
-----------+-----------+---------+------+------------------+-----------------------+-------------------------------------
 cn_5003   | localhost |   28811 | 8000 | /DWS/manager/app | /DWS/data1/coordinator | /DWS/manager/log/
Ruby/pg_log/cn_5003
(1 row)
```

pg_stat_get_thread()

描述：提供当前节点下所有线程的状态信息。

返回值类型：record

示例：

```
SELECT * FROM pg_stat_get_thread();
 node_name |       pid       |  lwpid  |    thread_name     |         creation_time
-----------+-----------------+---------+--------------------+-------------------------------
 cn_5003   | 281471515199536 |   28930 | JobScheduler       | 2023-01-04 07:03:16.086885+00
 cn_5003   | 281471498418224 |   28931 | StatCollector      | 2000-01-01 00:00:00+00
 cn_5003   | 281471464855600 |   28933 | WDRSnapshot        | 2023-01-04 07:03:16.086775+00
 cn_5003   | 281471380949040 |   28938 | WorkloadMonitor    | 2023-01-04 07:03:16.074454+00
 cn_5003   | 281471414511664 |   28936 | workload           | 2023-01-04 07:03:16.075457+00
 cn_5003   | 281471364167728 |   28939 | WLMArbiter         | 2023-01-04 07:03:16.076753+00
 cn_5003   | 281471397730352 |   28937 | CalculateSpaceInfo | 2023-01-04 07:03:16.078981+00
 cn_5003   | 281470777964592 | 1933534 | wlm                | 2023-01-13 08:01:32.350808+00
 cn_5003   | 281470889130032 | 1786064 | cn_5002            | 2023-01-13 07:01:50.173568+00
 cn_5003   | 281471299672112 |   29006 | cm_agent           | 2023-01-04 07:03:18.03415+00
 cn_5003   | 281471222065200 |   29970 | cn_5002            | 2023-01-04 07:03:39.694702+00
 cn_5003   | 281471238846512 | 1897367 | cn_5002            | 2023-01-04 20:01:40.611019+00
 cn_5003   | 281470905911344 |   30053 | cn_5002            | 2023-01-04 07:03:44.065774+00
 cn_5003   | 281470410537008 | 1933902 | cn_5002            | 2023-01-13 08:01:38.972574+00
 cn_5003   | 281470872348720 | 1880248 | cn_5001            | 2023-01-13 07:39:24.231418+00
 cn_5003   | 281471316453424 | 1883059 | cn_5001            | 2023-01-13 07:40:16.885667+00
 cn_5003   | 281470845081648 | 1305053 | cn_5001            | 2023-01-13 03:40:17.366784+00
 cn_5003   | 281470700357680 | 1500466 | wlm                | 2023-01-13 05:02:05.714544+00
 cn_5003   | 281470473455664 | 1883060 | cn_5001            | 2023-01-13 07:40:16.885963+00
 cn_5003   | 281470717138992 |   32065 | cm_agent           | 2023-01-04 07:04:23.906691+00
 cn_5003   | 281470807328816 | 1977925 | gsql               | 2023-01-13 08:20:04.509437+00
 cn_5003   | 281470683576368 | 1835242 | cn_5001            | 2023-01-13 07:20:16.549546+00
 cn_5003   | 281471584946224 |   28927 | Background writer  | 2023-01-04 07:03:16.065631+00
 cn_5003   | 281471633184816 |   28926 | CheckPointer       | 2023-01-04 07:03:16.065872+00
 cn_5003   | 281471548762160 |   28928 | Wal Writer         | 2023-01-04 07:03:16.066366+00
 cn_5003   | 281471448074288 |   28934 | TwoPhase Cleaner   | 2023-01-04 07:03:16.071172+00
 cn_5003   | 281471431292976 |   28935 | LWLock Monitor     | 2023-01-04 07:03:16.072897+00
 cn_5003   | 281470666795056 | 1210459 | CBM Writer         | 2023-01-04 15:16:05.543143+00
(28 rows)
```

pgxc_get_os_threads()

描述：提供当前集群中所有正常节点下的线程状态信息。

返回值类型：record

pg_stat_get_sql_count()

描述：提供当前节点中所有用户执行的SELECT/UPDATE/INSERT/DELETE/MERGE INTO语句的计数和响应时间的统计结果以及DDL、DML、DCL语句的计数结果。

返回值类型：record

示例：

```
SELECT * FROM pg_stat_get_sql_count();
 node_name |       user_name         | select_count | update_count | insert_count | delete_count |
mergeinto_count | ddl_count | dml_count | dcl_count | total_select_elapse | avg_select_elapse | max_select_el
apse | min_select_elapse | total_update_elapse | avg_update_elapse | max_update_elapse |
min_update_elapse | total_insert_elapse | avg_insert_elapse | max_insert_elapse | min_insert_elapse |
total_delete_ela
pse | avg_delete_elapse | max_delete_elapse | min_delete_elapse
-----------+-------------------------+--------------+--------------+--------------+--------------+-----------------
+-----------+-----------+-----------+--------------------+------------------+--------------
-----+------------------+--------------------+------------------+------------------+------------------
+---------------------+-------------------+-------------------+-------------------+-----------------
----+------------------+------------------+------------------
 cn_5003   | gs_role_read_all_stats |           0 |           0 |           0 |           0 |             0 |        0 |        0
|         0 |                 0 |               0 |
   0 |             0 |               0 |               0 |             0 |             0 |               0 |             0
|             0 |               0 |
  0 |             0 |               0 |               0
 cn_5003   | gs_role_signal_backend |           0 |           0 |           0 |           0 |             0 |        0 |        0
|         0 |                 0 |               0 |
   0 |             0 |               0 |               0 |             0 |             0 |               0 |             0
|             0 |               0 |
  0 |             0 |               0 |               0
 cn_5003   | gs_role_analyze_any    |           0 |           0 |           0 |           0 |             0 |        0 |        0
|         0 |                 0 |               0 |
   0 |             0 |               0 |               0 |             0 |             0 |               0 |             0
|             0 |               0 |
  0 |             0 |               0 |               0
 cn_5003   | gs_role_vacuum_any     |           0 |           0 |           0 |           0 |             0 |        0 |        0
|         0 |                 0 |               0 |
   0 |             0 |               0 |               0 |             0 |             0 |               0 |             0
|             0 |               0 |
  0 |             0 |               0 |               0
 cn_5003   | dbadmin                |         641 |           0 |           3 |           0 |             0 |       18 |      651
|         5 |           19236129 |            30009 |          814
0206 |             357 |                 0 |               0 |               0 |               0 |             70595 |
23531 |            62102 |             2750 |
  0 |             0 |               0 |               0
 cn_5003   | Ruby                   |      2078187 |        3263 |        22841 |           0 |             0 |     10436 |
2242517 |     16979 |         3753441293 |             1806 |           52
6891 |             191 |           67483165 |            20681 |            38076 |            15444 |          291598980
|            12766 |            35376 |             3791 |
  0 |             0 |               0 |               0
 cn_5003   | joe                    |           0 |           0 |           0 |           0 |             0 |        0 |        0 |
0 |             0 |               0 |
   0 |             0 |               0 |               0 |               0 |               0 |               0 |               0
|               0 |               0 |
  0 |             0 |               0 |               0
 cn_5003   | sea                    |         192 |           0 |           5 |           3 |             0 |        3 |      205
|         0 |            2561878 |            13343 |            68
1866 |             388 |                 0 |               0 |               0 |               0 |             11349 |
2269 |            3241 |             1521 |             19
255 |            6418 |            10656 |             2798
 cn_5003   | jj                     |           0 |           0 |           0 |           0 |             0 |        0 |        0 |        0
|               0 |               0 |
   0 |             0 |               0 |               0 |               0 |               0 |               0 |               0
|               0 |               0 |
  0 |             0 |               0 |               0
 cn_5003   | u1                     |           2 |           0 |           2 |           0 |             0 |        1 |        4 |
0 |              3712 |            1856 |
2407 |            1305 |                 0 |               0 |               0 |               0 |              5366 |
```

```
2683 |            3359 |            2007 |
  0 |               0 |               0 |               0
(10 rows)
```

pgxc_get_sql_count()

描述：提供当前集群所有节点中所有用户执行的SELECT/UPDATE/INSERT/DELETE/MERGE INTO语句的计数和响应时间的统计结果以及DDL、DML、DCL语句的计数结果。

返回值类型：record

pgxc_get_workload_sql_count()

描述：提供当前集群所有CN上所有Workload控制组内执行的SELECT/UPDATE/INSERT/DELETE语句的计数统计结果以及DDL、DML、DCL类型语句的计数统计结果。

返回值类型：record

示例：

```
SELECT * FROM pgxc_get_workload_sql_count();
 node_name |    workload    | select_count | update_count | insert_count | delete_count | ddl_count |
dml_count | dcl_count
-----------+----------------+--------------+--------------+--------------+--------------+-----------+-----------+-----------
 cn_5003   | default_pool |      2079352 |         3264 |        22858 |            3 |     10460 |   2243738 |     16988
 cn_5001   | default_pool |      2201345 |            9 |            0 |            0 |     10474 |   2359633 |     10465
 cn_5002   | default_pool |      3784696 |            0 |       103106 |          136 |     10438 |   4039090 |     10498
(3 rows)
```

pgxc_get_workload_sql_elapse_time()

描述：提供当前集群所有CN上所有Workload控制组内执行的SELECT/UPDATE/INSERT/DELETE语句的响应时间统计结果。

返回值类型：record

示例：

```
SELECT * FROM pgxc_get_workload_sql_elapse_time();
 node_name |    workload    | total_select_elapse | max_select_elapse | min_select_elapse | avg_select_elapse |
total_update_elapse | max_update_elapse | min_update_elapse | avg_update_elapse | total_insert_el
apse | max_insert_elapse | min_insert_elapse | avg_insert_elapse | total_delete_elapse | max_delete_elapse |
min_delete_elapse | avg_delete_elapse
-----------+----------------+---------------------+-------------------+-------------------+-------------------
+---------------------+-------------------+-------------------+-------------------+----------------
-----+-------------------+-------------------+-------------------+---------------------+-------------------
+-------------------+-------------------
 cn_5003   | default_pool |          3776420502 |           8140206 |                 0 |              1816 |
67505332 |             38076 |                 0 |             20682 |           29178
9830 |             62102 |                 0 |             12765 |              19255 |             10656 |                 0
|              6418
 cn_5001   | default_pool |          8599339496 |           3390159 |                 0 |              3906 |
52789 |             18207 |                 0 |              5865 |
   0 |                0 |                 0 |                 0 |                   0 |                 0 |                0 |                 0
 cn_5002   | default_pool |         40483096221 |           2178781 |                 0 |             10695 |
0 |                0 |                 0 |                 0 |          13310238
8148 |           2398854 |                 0 |           1290752 |             2072031 |             52877 |                 0
|             15236
(3 rows)
```

get_instr_unique_sql()

描述：提供当前节点中收集的Unique SQL的信息。如果是CN节点，将返回该CN上收集的Unique SQL的完整信息，即会收集其他CN和DN上对应Unique SQL的信息并进行汇总展示；如果是DN节点，将返回本DN节点上的Unique SQL信息。详见视图GS_INSTR_UNIQUE_SQL。

返回值类型：record

reset_instr_unique_sql(cstring, cstring, INT8)

描述：清理已收集的Unique SQL信息。输入参数含义如下：

- GLOBAL/LOCAL：清理范围为所有节点或当前节点。
- ALL/BY_USERID/BY_CNID/BY_GUC：ALL表示清理所有，BY_USERID/BY_CNID表示按照USERID或CNID进行清理，BY_GUC表示清理操作是由GUC参数instr_unique_sql_count设置值变小引起的。
- 第三个参数值对应第二个参数设置，ALL/BY_GUC的情况下该值无意义。

返回值类型：bool

pgxc_get_instr_unique_sql()

描述：提供集群中所有CN上收集的Unique SQL的完整信息。该函数只能在CN上执行。

返回值类型：record

get_instr_unique_sql_remote_cns()

描述：提供集群中除正在执行此函数的CN之外的所有CN上收集的Unique SQL的完整信息。该函数只能在CN上执行。

返回值类型：record

pgxc_get_node_env()

描述：提供获取集群中所有节点的环境变量信息。

返回值类型：record

示例：

```
SELECT * FROM pgxc_get_node_env();
  node_name   |     host      | process | port |   installpath    |        datapath         |           log_directory
--------------+---------------+---------+------+------------------+-------------------------+-----------------------------------
 dn_6001_6002 | 172.16.102.5  |   24443 | 40000 | /DWS/manager/app | /DWS/data1/h0dn1/primary0 | /DWS/manager/log/Ruby/pg_log/dn_6001
 dn_6003_6004 | 172.16.70.17  |   21823 | 40000 | /DWS/manager/app | /DWS/data1/h1dn1/primary0 | /DWS/manager/log/Ruby/pg_log/dn_6003
 dn_6005_6006 | 172.16.120.50 |   22331 | 40000 | /DWS/manager/app | /DWS/data1/h2dn1/primary0 | /DWS/manager/log/Ruby/pg_log/dn_6005
 cn_5003      | localhost     |   28811 |  8000 | /DWS/manager/app | /DWS/data1/coordinator    | /DWS/manager/log/Ruby/pg_log/cn_5003
 cn_5001      | 172.16.102.5  |   30873 |  8000 | /DWS/manager/app | /DWS/data1/coordinator    | /DWS/manager/log/Ruby/pg_log/cn_5001
 cn_5002      | 172.16.70.17  |   29229 |  8000 | /DWS/manager/app | /DWS/data1/coordinator    | /DWS/manager/log/Ruby/pg_log/cn_5002
(6 rows)
```


pv_compute_pool_workload()

描述：提供云上加速集群当前负载信息。

返回值类型：record

pg_stat_get_status(tid, num_node_display)

描述：查询当前实例中工作线程（backend thread）以及辅助线程（auxiliary thread）的阻塞等待情况，其返回结果的详细含义参见PG_THREAD_WAIT_STATUS视图。输入参数含义如下：

- tid：表示线程ID，bigint类型。如果为NULL，则返回所有工作线程和辅助线程的等待情况；否则只返回指定ID线程的等待情况。
- num_node_display：integer类型。对于等待状态为“wait node”的记录，指定其wait_status列中显示的被等待节点的最大数量。
  - 如果为空或者小于等于0，则只显示一个被等待节点。
  - 如果大于20，则最多只显示20个节点。
  - 如果大于0且小于等于20，则显示数量为num_node_display和实际被等待节点数量的最小者。例如查询“SELECT * from pg_stat_get_status(NULL, 10)”，如果实际被等待节点数量大于10，则只随机显示其中10个节点名称，如果实际被等待节点数量小于等于10，则显示全部被等待节点名称。当实际被等待节点数量大于显示数量时，被显示的节点名称为随机挑选。

返回值类型：record

pgxc_get_thread_wait_status(num_node_display)

描述：查询集群各个节点上所有SQL语句产生的线程之间的调用层次关系，以及各个线程的阻塞等待状态。其返回结果的详细含义参见PGXC_THREAD_WAIT_STATUS视图。输入参数num_node_display的类型和含义与上述函数pg_stat_get_status相同。

返回值类型：record

pgxc_os_run_info()

描述：查询集群中各节点上操作系统运行的状态信息。函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PV_OS_RUN_INFO”章节。

返回值类型：record

get_instr_wait_event()

描述：查询当前实例上各类等待状态和事件的统计信息。函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>GS_WAIT_EVENTS”。如果GUC参数enable_track_wait_event为off，则返回0行。

返回值类型：record

pgxc_wait_events()

描述：查询集群中各节点上各类等待状态和事件的统计信息，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PGXC_WAIT_EVENTS”视图。如果GUC参数enable_track_wait_event为off，则返回0行。

返回值类型：record

pgxc_stat_bgwriter()

描述：查询集群中各节点上后端写进程活动的统计信息，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PG_STAT_BGWRITER”视图。

返回值类型：record

示例：

```
SELECT * FROM pgxc_stat_bgwriter();
  node_name   | checkpoints_timed | checkpoints_req | checkpoint_write_time | checkpoint_sync_time |
buffers_checkpoint | buffers_clean | maxwritten_clean | buffers_backend | buffers_backend_fsync | buffers_
alloc |       stats_reset
--------------+-------------------+-----------------+-----------------------+----------------------+--------------------
+---------------+------------------+-----------------+-----------------------+---------
------+------------------------
 dn_6001_6002 |                 0 |               0 |                     0 |                    0 |               0 |           0
|               0 |               0 |                   0 |
    0 | 2000-01-01 00:00:00+00
 dn_6003_6004 |                 0 |               0 |                     0 |                    0 |               0 |           0
|               0 |               0 |                   0 |
    0 | 2000-01-01 00:00:00+00
 dn_6005_6006 |                 0 |               0 |                     0 |                    0 |               0 |           0
|               0 |               0 |                   0 |
    0 | 2000-01-01 00:00:00+00
 cn_5003      |                 0 |               0 |                     0 |                    0 |               0 |           0
|               0 |               0 |                   0 |
    0 | 2000-01-01 00:00:00+00
 cn_5001      |                 0 |               0 |                     0 |                    0 |               0 |           0
|               0 |               0 |                   0 |
    0 | 2000-01-01 00:00:00+00
 cn_5002      |                 0 |               0 |                     0 |                    0 |               0 |           0
|               0 |               0 |                   0 |
    0 | 2000-01-01 00:00:00+00
(6 rows)
```

pgxc_stat_replication()

描述：查询集群中各节点上日志同步的状态信息，如发起端发送日志位置，收端接收日志位置等，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PG_STAT_REPLICATION”视图。

返回值类型：record

示例：

```
SELECT * FROM pgxc_stat_replication();
  node_name   |       pid       | usesysid | usename |     application_name     |  client_addr  | client_hostname |
client_port |         backend_start         |   state   | sender_sent_location | receiver_wri
te_location | receiver_flush_location | receiver_replay_location | sync_priority | sync_state
--------------+-----------------+----------+---------+--------------------------+---------------+-----------------
+-------------+-------------------------------+-----------+----------------------+-------------
------------+-------------------------+--------------------------+---------------+------------
 dn_6001_6002 | 281469637695536 |       10 | Ruby    | WalSender to Standby     | 172.16.70.17  |
|       26084 | 2023-01-04 07:01:27.348647+00 | Streaming | 0/4940D6B8           | 0/4940D6B8
           | 0/4940D6B8             | 0/4940D6B8              |             1 | Sync
 dn_6001_6002 | 281469304735792 |       10 | Ruby    | WalSender to Secondary | 172.16.120.50 |
|       35214 | 2023-01-04 07:01:29.51929+00  | Streaming | 0/4000000            | 0/4000000
           | 0/4000000              | 0/4000000               |             0 | Sync
 dn_6003_6004 | 281469634050096 |       10 | Ruby    | WalSender to Standby     | 172.16.120.50 |
|       13072 | 2023-01-04 07:01:26.28706+00  | Streaming | 0/493EF000           | 0/493EF000
           | 0/493EF000             | 0/493EF000              |             1 | Sync
 dn_6003_6004 | 281469563295792 |       10 | Ruby    | WalSender to Secondary | 172.16.102.5  |
```

```
|       55068 | 2023-01-04 07:01:29.310595+00 | Streaming | 0/4000000           | 0/4000000
           | 0/4000000             | 0/4000000             |             0 | Sync
 dn_6005_6006 | 281470349690928 |       10 | Ruby    | WalSender to Standby   | 172.16.102.5  |
|       40376 | 2023-01-04 07:01:26.768434+00 | Streaming | 0/49415A70          | 0/49415A70
           | 0/49415A70            | 0/49415A70            |             1 | Sync
 dn_6005_6006 | 281470010435632 |       10 | Ruby    | WalSender to Secondary | 172.16.70.17  |
|       33750 | 2023-01-04 07:01:29.499269+00 | Streaming | 0/4000000           | 0/4000000
           | 0/4000000             | 0/4000000             |             0 | Sync
(6 rows)
```

pgxc_replication_slots()

描述：查询集群中各DN上复制的状态信息，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PG_REPLICATION_SLOTS”视图。

返回值类型：record

示例：

```
SELECT * FROM pgxc_replication_slots();
  node_name   |    slot_name    | plugin | slot_type | datoid | database | active | x_min | catalog_xmin |
restart_lsn    | dummy_standby
--------------+-----------------+--------+-----------+--------+----------+--------+-------+--------------
+------------------+---------------
 dn_6001_6002 | dn_6001_3002    |        | physical  |      0 |          | t      |       |              |                  | t
 dn_6001_6002 | dn_6001_6002    |        | physical  |      0 |          | t      |       |              | 0/49448720       | f
 dn_6001_6002 | gs_roach_common |        | physical  |      0 |          | f      |       |    635143447 | FFFFFFFF/
FFFFFFFF | f
 dn_6003_6004 | dn_6003_3003    |        | physical  |      0 |          | t      |       |              |                  | t
 dn_6003_6004 | dn_6003_6004    |        | physical  |      0 |          | t      |       |              | 0/4942B760       | f
 dn_6003_6004 | gs_roach_common |        | physical  |      0 |          | f      |       |    634883623 | FFFFFFFF/
FFFFFFFF | f
 dn_6005_6006 | dn_6005_3004    |        | physical  |      0 |          | t      |       |              |                  | t
 dn_6005_6006 | dn_6005_6006    |        | physical  |      0 |          | t      |       |              | 0/4944CE80       | f
 dn_6005_6006 | gs_roach_common |        | physical  |      0 |          | f      |       |    635285455 | FFFFFFFF/
FFFFFFFF | f
(9 rows)
```

pgxc_settings()

描述：查询集群中各节点上运行时参数的相关信息，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PG_SETTINGS”视图。

返回值类型：record

pgxc_instance_time()

描述：查询集群中各节点的运行时间统计信息及各执行阶段所消耗时间，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PV_INSTANCE_TIME”视图。

返回值类型：record

pg_stat_get_redo_stat()

描述：查询当前节点上的XLOG重做统计信息，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PV_REDO_STAT”视图。

返回值类型：record

示例：

```
SELECT * FROM pg_stat_get_redo_stat();
 phywrts | phyblkwrt | writetim | avgiotim | lstiotim | miniotim | maxiowtm
```



```
---------+-----------+----------+----------+----------+----------+----------
 400171 |    552783 | 11040710 |       27 |       20 |        8 |     7401
(1 row)
```

pgxc_redo_stat()

描述：查询集群中各节点上的XLOG重做统计信息，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>PV_REDO_STAT”视图。

返回值类型：record

示例：

```
SELECT * FROM  pgxc_redo_stat();
  node_name   | phywrts | phyblkwrt | writetim | avgiotim | lstiotim | miniotim | maxiowtm
--------------+---------+-----------+----------+----------+----------+----------+----------
 dn_6001_6002 |  698244 |    836088 | 17608019 |       25 |       15 |        8 |    13115
 dn_6003_6004 |  661128 |    799636 | 16714302 |       25 |       21 |        8 |     8195
 dn_6005_6006 |  698146 |    836178 | 18117951 |       25 |       24 |        8 |     8326
 cn_5003      |  400206 |    552823 | 11041701 |       27 |       18 |        8 |     7401
 cn_5001      |  380931 |    514233 | 10174114 |       26 |       19 |        8 |     7726
 cn_5002      |  551727 |    687991 | 11859292 |       21 |       31 |        8 |    10310
(6 rows)
```

get_local_rel_iostat()

描述：查询当前实例上磁盘读写的统计信息。函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>GS_REL_IOSTAT”视图。

返回值类型：record

pgxc_rel_iostat()

描述：查询集群中各节点上磁盘读写的统计信息，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>GS_REL_IOSTAT”视图。

返回值类型：record

get_node_stat_reset_time()

描述：返回当前实例统计信息被重置的时间。

返回值类型：timestamptz

pgxc_node_stat_reset_time()

描述：查询集群中各节点统计信息被重置的时间，函数返回结果信息详见《开发指南》中“系统表和系统视图>系统视图>GS_NODE_STAT_RESET_TIME”视图。

返回值类型：record

说明

实例正常运行过程中，内存中的各类统计数值会逐渐累加，以下情况会导致内存中的统计数值被重置为0：

- 实例重启或集群发生了切换；
- 数据库Database被删除（drop）；
- 用户执行了重置操作，如执行pgstat_recv_resetcounter函数会将数据库中的统计计数器清零，执行reset_instr_unique_sql函数会将Unique SQL数据清零。

如果发生了以上事件，GaussDB(DWS)会记录统计信息被重置的时间，可通过get_node_stat_reset_time函数查询。

pgxc_parallel_query(text, text)

描述：在指定类型的数据实例上执行指定的SQL查询语句，并把查询语句的结果返回给当前CN。该函数8.1.2及以上版本支持。

函数有两个参数：

第一个参数：SQL语句在哪些实例上执行。当前支持的有效入参是 'dn'，'datanode'，'cn'， 'coordinator'， 'all'， 其它值会导致函数执行报错。其中 'dn'，'datanode'表示在所有DN上执行，'cn'， 'coordinator'表示在所有CN上执行， all表示在所有的CN和所有的DN上执行。

第二个参数：需要发往远程节点执行的SQL语句，函数内部会对SQL语句中查询的对象进行校验，不支持用户表、分布式表和自定义的多结果集函数。

返回值类型：record

说明

- 该函数功能仅为便于开发人员高效的收集集群内实例执行信息或者状态的视图，不建议用户直接使用。
- 该函数为多结果集函数，返回的数据类型为record，所以需要在函数调用后面增加AS语句显示的指出输出的列名和数据类型，如下所示：
  ```
  SELECT * FROM pgxc_parallel_query('all', 'select node_name, db_name, thread_name, query_id, tid, lwtid, ptid, tlevel, smpid, wait_status, wait_event from pg_thread_wait_status') AS (node_name text, db_name text, thread_name text, query_id bigint, tid bigint, lwtid integer , ptid integer, tlevel integer , smpid integer, wait_status text, wait_event text);
  ```
- 函数第二个参数指定的SQL语句输出结果的数据类型必须跟AS后面指明的数据类型一致，否则执行时可能会因为类型不匹配而报错。
- 函数第二个参数指定的SQL语句中不能触发跨节点的查询动作，否则会触发执行报错。
- 函数第二个参数指定的SQL语句只能是SELECT/UPDATE/DELETE/INSERT语句中的一种，且
  - 不支持语句中有returning语句。
  - 函数调用用户需具有SQL语句中对象相应的操作权限。
  - 如果是INSERT语句，不支持INSERT OVERWRITE、UPSERT、和INSERT INTO。
  - 对于UPDATE/DELETE/INSERT语句，只允许初始化用户在就地升级模式下或者管理员用户在重分布模式下执行；要求语句在每个实例上修改的记录数相同，否则执行会报错；函数会输出一条一列bigint类型的数值，此数值表示语句在每个实例上操作的记录数。
    ```
    SELECT * FROM pgxc_parallel_query('cn', 'UPDATE pg_partition SET relpages = 0') AS (updated bigint);
    ```


generate_wdr_report(begin_snap_id bigint, end_snap_id bigint, report_type cstring, report_scope cstring, node_name cstring)

描述：创建负荷分析报告。输入参数含义如下：

- begin_snap_id，end_snap_id：生成报告的起止快照ID，bigint型，要求begin_snap_id < end_snap_id，并且起止快照的时间没有交集。判断快照时间是否有交集可通过查询dbms_om.snapshot表格select s1.end_ts < s2.start_ts from (select * from dbms_om.snapshot where snapshot_id=$begin_snap_id) as s1, (select * from dbms_om.snapshot where snapshot_id=$end_snap_id) as s2;，如果返回true则没有交集，反之有交集。
- report_type：报告类型，cstring型，包括“summary”，“detail”，“all”三种类型。
- report_scope：报告范围，cstring型，包括“cluster”和“node”两种。
- node_name：节点名称，cstring型，如果report_scope="node"，该参数必须是pg_catalog. pgxc_node表格中node_name字段里的CN或DN节点名称。

返回值类型：text

说明

- 该函数只有数据库管理员SYSADMIN才有权执行，非管理员执行会提示无权限。
- 该函数只能在CN上执行，在DN上执行会提示错误：“WDR report can only be created on coordinator.”。
- 如果生成报告成功，会返回：“Report %s has been generated”。
- 生成报告的两个快照期间不能发生统计信息重置事件，否则会提示错误：“Instance reset time is different”。引起统计信息重置的事件参见**pgxc_node_stat_reset_time**函数。

wdr_xdb_query(db_name text, snapshot_id bigint, view_name text)

描述：查询指定数据库下的指定视图。有的视图在不同数据库中查询结果不同，例如global_table_stat视图用于查询表格的统计信息，由于不同数据库下表格不同，在不同数据库中查询该视图得到的结果也不同。wdr_xdb_query函数可以在当前连接中访问db_name指定的数据库，并在该数据库中查询view_name指定的视图。输入参数含义如下：

- db_name：指定的数据库名称，text型。
- snapshot_id：快照ID，bigint型，参见“性能视图快照”。
- view_name：指定视图名称，text型。视图名称必须在如下白名单中：
  - global_table_stat
  - global_table_change_stat
  - global_column_table_io_stat
  - global_row_table_io_stat

    返回值类型：record，其第一列为snapshot_id bigint，第二列为db_name text，其他列的名称、类型和顺序与view_name指定的视图相同。

    示例：

```
SELECT snapshot_id, db_name, schemaname, relname, distribute_mode,
seq_scan ,seq_tuple_read ,index_scan ,index_tuple_read ,tuple_inserted
,tuple_updated ,tuple_deleted ,tuple_hot_updated ,live_tuples ,dead_tuples from
wdr_xdb_query('postgres'::text, 1, 'global_table_stat'::text) as i(snapshot_id bigint, db_name text,
schemaname name, relname name, distribute_mode char, seq_scan bigint, seq_tuple_read
```

```
bigint, index_scan bigint, index_tuple_read bigint, tuple_inserted bigint, tuple_updated bigint,
tuplee_deleted bigint, tuple_hot_updated bigint, live_tuples bigint, dead_tuples bigint);
```

说明

- 该函数仅8.1.2及以上版本支持。
- 该函数只有数据库管理员SYSADMIN才有权执行，非管理员执行会提示无权限。
- 该函数只能查询白名单中的视图，如果查询其他视图，会报错：“Input view name is invalid.”。

vac_fileclear_relation(oid)

描述：用于强制清理指定列存表中被VACUUM重写的文件，完成空间回收。

参数：列存表oid。

返回值类型：integer

说明

- 使用本函数前需要设置参数colvacuum_threshold_scale_factor，并确保VACUUM对指定列存表的文件完成重写后，才会清理文件并回收空间。
- 本函数将对指定的列存表施加排他锁。

vac_fileclear_all_relation()

描述：用于强制清理所有列存表中被VACUUM重写的文件，完成空间回收。

返回值类型：record

6.28 数据库对象函数

6.28.1 数据库对象尺寸函数

数据库对象尺寸函数计算数据库对象使用的实际磁盘空间。

pg_column_size(any)

描述：存储一个指定的数值需要的字节数（可能压缩过）。

返回值类型：integer

备注：pg_column_size显示用于存储某个独立数据值的空间。

```
SELECT pg_column_size(1);
 pg_column_size
----------------
              4
(1 row)
```

pg_database_size(oid)

描述：指定OID代表的数据库使用的磁盘空间。

返回值类型：bigint

pg_database_size(name)

描述：指定名称的数据库使用的磁盘空间。

返回值类型：bigint

备注：pg_database_size接收一个数据库的OID或者名字，然后返回该对象使用的全部磁盘空间。

示例：

```
SELECT pg_database_size('gaussdb');
 pg_database_size
------------------
         51590112
(1 row)
```

pg_relation_size(oid)

描述：指定OID代表的表或者索引所使用的磁盘空间。

返回值类型：bigint

get_db_source_datasize()

描述：估算当前数据库非压缩态的数据总容量。

返回值类型：bigint

备注：（1）调用该函数前需要做analyze；（2）通过估算列存的压缩率计算非压缩态的数据总容量。

示例：

```
analyze;
ANALYZE
SELECT get_db_source_datasize();
 get_db_source_datasize
------------------------
            35384925667
(1 row)
```

pg_relation_size(text)

描述：指定名称的表或者索引使用的磁盘空间。表名字可以用模式名修饰。

返回值类型：bigint

pg_relation_size(relation regclass, fork text)

描述：指定表或索引的指定分叉树（'main'，'fsm'或'vm'）使用的磁盘空间。

返回值类型：bigint

pg_relation_size(relation regclass)

描述：pg_relation_size(..., 'main')的简写。

返回值类型：bigint

备注：pg_relation_size接受一个表、索引、压缩表的OID或者名字，然后返回它们的字节大小。

pg_partition_size(oid,oid)

描述：指定OID代表的分区使用的磁盘空间。其中，第一个oid为表的OID，第二个oid为分区的OID。

返回值类型：bigint

pg_partition_size(text, text)

描述：指定名称的分区使用的磁盘空间。其中，第一个text为表名，第二个text为分区名。

返回值类型：bigint

pg_partition_indexes_size(oid,oid)

描述：指定OID代表的分区的索引使用的磁盘空间。其中，第一个oid为表的OID，第二个oid为分区的OID。

返回值类型：bigint

pg_partition_indexes_size(text,text)

描述：指定名称的分区的索引使用的磁盘空间。其中，第一个text为表名，第二个text为分区名。

返回值类型：bigint

pg_indexes_size(regclass)

描述：附加到指定表的索引使用的总磁盘空间。

返回值类型：bigint

pg_size_pretty(bigint)

描述：把字节计算的尺寸转换成一个易读的尺寸。

返回值类型：text

pg_size_pretty(numeric)

描述：把用数值表示的字节计算的尺寸转换成一个易读的尺寸。

返回值类型：text

备注：pg_size_pretty用于把其他函数的结果格式化成一种易读的格式，可以根据情况使用KB 、MB 、GB 、TB。

pg_table_size(regclass)

描述：指定的表使用的磁盘空间，不计索引（但是包含TOAST，自由空间映射和可见性映射）。

返回值类型：bigint

pg_total_relation_size(oid)

描述：指定OID代表的表使用的磁盘空间，包括索引和压缩数据。

返回值类型：bigint

pg_total_relation_size(regclass)

描述：指定的表使用的总磁盘空间，包括所有的索引和TOAST数据。

返回值类型：bigint

pg_total_relation_size(text)

描述：指定名字的表所使用的全部磁盘空间，包括索引和压缩数据。表名字可以用模式名修饰。

返回值类型：bigint

备注：pg_total_relation_size接受一个表或者一个压缩表的OID或者名称，然后返回以字节计的数据和所有相关的索引和压缩表的尺寸。

6.28.2 数据库对象位置函数

pg_relation_filenode(relation regclass)

描述：指定关系的文件节点数。

返回值类型：oid

备注：pg_relation_filenode接收一个表、索引、序列或压缩表的OID或者名字，并且返回当前分配给它的“filenode”数。文件节点是关系使用的文件名字的基本组件。对大多数表来说，结果和pg_class.relfilenode相同，但对确定的系统目录来说，relfilenode为0而且这个函数必须用来获取正确的值。如果传递一个没有存储的关系，比如一个视图，那么这个函数返回NULL。

pg_relation_filepath(relation regclass)

描述：指定关系的文件路径名。

返回值类型：text

备注：pg_relation_filepath类似于pg_relation_filenode，但是它返回关系的整个文件路径名（相对于数据库集群的数据目录PGDATA）。

6.28.3 分区管理函数

proc_add_partition (relname regclass, boundaries_interval interval)

描述：用于给开启自动创建分区功能的表添加分区。

返回值类型：void

备注：该函数运行时，会在现有分区boundary的基础上，创建多个时间范围为boundaries_interval的新分区，直到new_part_boundary – now_time >= 29 *

boundaries_interval，之后再额外多创建一个分区，保证该函数运行时，一定会创建一个新分区 。

示例：

```
call proc_add_partition('my_schema.my_table', interval '1 day');
 proc_add_partition
--------------------

(1 row)
```

proc_drop_partition (relname regclass, older_than interval)

描述：用于给开启自动删除分区功能的表删除分区。

返回值类型：void

备注：该函数运行时，遍历表所有分区，并删除其中boundary小于(now_time - older_than)的分区；如果所有分区都满足删除条件，则保留一个分区，并truncate该表。

示例：

```
call proc_drop_partition('my_schema.my_table', interval '7 day');
 proc_drop_partition
--------------------

(1 row)
```

6.28.4 排序规则版本函数

pg_collation_actual_version (oid)

描述：返回当前安装在操作系统中的该排序规则对象的实际版本，目前仅对case_insensitive有效。

返回值类型：text

示例：

```
SELECT oid FROM pg_collation WHERE collname ='case_insensitive';
 oid
------
 3300
(1 row)

SELECT pg_collation_actual_version(3300);
 pg_collation_actual_version
-----------------------------
 153.14
(1 row)
```

6.28.5 冷热表用户函数

pg_obs_cold_refresh_time(table_name, time)

描述：用来修改冷热表的冷数据切换至OBS上的时间，默认为每日0点。

table_name为冷热表表名，类型为name，time为数据切换任务调度时间，类型为Time。

返回值：SUCCESS，任务时间修改成功。

示例：

```
SELECT * FROM pg_obs_cold_refresh_time('lifecycle_table', '06:30:00');
pg_obs_cold_refresh_time
--------------------------
 SUCCESS
(1 row)
```

pg_refresh_storage()

描述：切换所有冷热表，将符合冷热切换规则的数据由热数据切换至冷数据（OBS中）。

返回值类型：int

返回值字段：

- success_count int：切换成功的表个数
- failed_count int：切换失败的表个数

示例：

```
SELECT * FROM pg_refresh_storage();
 success_count | failed_count
---------------+--------------
             1 |            0
(1 row)
```

pg_lifecycle_table_data_distribute(table_name)

描述：查看某个冷热表的数据分布情况。

table_name为表名，不可缺省。

返回值：record

示例：根据节点数量形成多条记录，如下示例为只有一个dn节点时w1表数据分布情况。

```
SELECT * FROM pg_catalog.pg_lifecycle_table_data_distribute('w1');
 schemaname | tablename | nodename | hotpartition | coldpartition | switchablepartition | hotdatasize |
colddatasize | switchabledatasize
------------+-----------+----------+--------------+---------------+---------------------+-------------+--------------
+---------------------
 public     | w1        | dn_1     | p2           | p1            |                     | 80 KB       | 0 bytes      | 0 bytes
(1 row)
```

pg_lifecycle_node_data_distribute()

描述：查看所有冷热表数据分布情况。

返回值：record

示例：数据库中当前存在两个冷热表，其数据分布情况如下。

```
SELECT * FROM pg_catalog.pg_lifecycle_node_data_distribute();
 schemaname | tablename | nodename | hotpartition | coldpartition | switchablepartition | hotdatasize |
colddatasize | switchabledatasize
------------+-----------+----------+--------------+---------------+---------------------+-------------+--------------
+---------------------
 public     | w1        | dn_1     | p2           | p1            |                     |       81920 |            0 |                    0
 public     | w2        | dn_1     | p2           | p1            |                     |       81920 |            0 |                    0
(2 rows)
```

6.29 残留文件管理函数

6.29.1 获取残留文件列表函数

pg_get_residualfiles()

描述：用于获取当前节点的所有残留文件记录。该函数为实例级函数，与当前所在的数据库无关，可以在任意实例上运行。

参数类型：无

返回值类型：record

函数返回字段如下：

表 6-30 pg_get_residualfiles()返回字段

名称	类型	描述
isverified	bool	是否已经验证。
isdeleted	bool	是否已经被删除。
dbname	text	所属数据库名称。
residualfile	text	数据文件路径。
filepath	text	残留文件记录路径。
notes	text	注释。

示例：

```
SELECT * FROM pg_get_residualfiles();
 isverified | isdeleted | dbname |   residualfile    |         filepath          | notes
------------+-----------+--------+-------------------+---------------------------+-------
 f          | f         | db2    | base/49155/114691 | pgrf_20200908160211441546 |
 f          | f         | db2    | base/49155/114694 | pgrf_20200908160211441546 |
 f          | f         | db2    | base/49155/114696 | pgrf_20200908160211441546 |
(3 rows)
```

pgxc_get_residualfiles()

描述：pg_get_residualfiles()的CN统一查询函数。该函数为集群级函数，与当前所在的数据库无关，在CN实例上运行。

参数类型：无

返回值类型：record

函数返回字段如下：

表 6-31 pgxc_get_residualfiles()返回字段

名称	类型	描述
nodename	text	节点名称。
isverified	bool	是否已经验证。
isdeleted	bool	是否已经被删除。
dbname	text	所属数据库名称。
residualfile	text	数据文件路径。
filepath	text	残留文件记录路径。
notes	text	注释。

示例：

```
SELECT * FROM pgxc_get_residualfiles();
   nodename   | isverified | isdeleted |  dbname  |    residualfile    |          filepath          | notes
--------------+------------+-----------+----------+--------------------+----------------------------+-------
 cn_5001      | f          | f         | postgres | base/15092/32803   | pgrf_20200910170129360401 |
 dn_6001_6002 | f          | f         | db2      | base/49155/114691 | pgrf_20200908160211441546 |
 dn_6001_6002 | f          | f         | db2      | base/49155/114694 | pgrf_20200908160211441546 |
 dn_6001_6002 | f          | f         | db2      | base/49155/114696 | pgrf_20200908160211441546 |
(4 rows)
```

6.29.2 验证残留文件函数

说明

pgxc类残留文件管理函数只对CN和当前主DN进行操作，不会验证和清理备DN上的残留文件。所以主DN完成清理后，应在备DN上及时执行残留文件清理操作或对备机进行build，防止主备切换后由于增量build导致备机残留文件被重新复制回主DN，导致未成功清理的假象。

pg_verify_residualfiles(filepath)

描述：用于验证参数指定文件中记录的文件是否为残留文件。该函数为实例级函数，与当前所在的数据库相关，可以在任意实例上运行。

参数类型：text

返回值类型：bool

函数返回字段如下：

表 6-32 pg_verify_residualfiles(filepath)返回字段

名称	类型	描述
isverified	bool	是否完成验证

示例：


```
SELECT * FROM pg_verify_residualfiles('pgrf_20200908160211441546');
 isverified
------------
 t
(1 row)
```

说明

本函数只能验证记录的文件在当前登录的数据库中是否是残留文件。如果记录的文件不属于当前登录的数据库，则不会进行校验行为。

pg_verify_residualfiles()

描述：用于验证当前实例上所有残留文件列表中记录的文件是否为残留文件。该函数为实例级函数，与当前所在的数据库相关，可以在任意实例上运行。

参数类型：无

返回值类型：record

函数返回字段如下：

表 6-33 pg_verify_residualfiles()返回字段

名称	类型	描述
result	bool	是否完成验证
filepath	text	残留文件记录路径
notes	text	注释

示例：

```
SELECT * FROM pg_verify_residualfiles();
 result |          filepath          | notes
--------+----------------------------+-------
 t      | pgrf_20200908160211441546 |
(1 row)
```

说明

本函数只能验证记录的文件在当前登录的数据库中是否是残留文件。如果记录的文件不属于当前登录的数据库，则不会进行校验行为。

pgxc_verify_residualfiles()

描述：pg_verify_residualfiles()的CN统一查询函数。该函数为集群级函数，与当前所在的数据库相关，在CN实例上运行。

参数类型：无

返回值类型：record

函数返回字段如下：

表 6-34 pgxc_verify_residualfiles()返回字段

名称	类型	描述
nodename	text	节点名称
result	bool	是否完成验证
filepath	text	残留文件记录路径
notes	text	注释

示例：

```
SELECT * FROM pgxc_verify_residualfiles();
   nodename   | result |          filepath           | notes
--------------+--------+-----------------------------+-------
 cn_5001      | t      | pgrf_20200910170129360401 |
 dn_6001_6002 | t      | pgrf_20200908160211441546 |
(2 rows)
```

说明

本函数只能验证记录的文件在当前登录的数据库中是否是残留文件。如果记录的文件不属于当前登录的数据库，则不会进行校验行为。

pg_is_residualfiles(residualfile)

描述：用于查询当前库中指定的relfilenode是否为残留文件。该函数为实例级函数，与当前所在的数据库相关，可以在任意实例上运行。

参数类型：text

返回值类型：bool

函数返回字段如下：

表 6-35 pg_is_residualfiles(residualfile)返回字段

名称	类型	描述
result	bool	是否是残留文件

示例：

```
SELECT * FROM pg_is_residualfiles('base/49155/114691');
 result
--------
 t
(1 row)
```


说明

本函数只能验证记录的文件在当前登录的database中是否为残留文件。如果记录的文件不属于当前登录的数据库，则会被检测为是残留文件。

例如：针对gaussdb数据库中的非残留文件 base/15092/14790，如果在gaussdb库中查询，则认为是非残留文件；在其他数据库中查询，则认为是残留文件。

```
SELECT * FROM pg_is_residualfiles('base/15092/14790');
result
--------
f
(1 row)

\c db2
db2=# SELECT * FROM pg_is_residualfiles('base/15092/14790');
result
--------
t
(1 row)
```

6.29.3 删除残留文件函数

说明

pgxc类残留文件管理函数只对CN和当前主DN进行操作，不会验证和清理备DN上的残留文件。所以主DN完成清理后，应在备DN上及时执行残留文件清理操作或对备机进行build，防止主备切换后由于增量build导致备机残留文件被重新复制回主DN，导致未成功清理的假象。

pg_rm_residualfiles(filepath)

描述：用于删除当前实例中指定残留文件列表中的文件。该函数为实例级函数，与当前所在的数据库无关，可以在任意实例上运行。

参数类型：text

返回值类型：record

函数返回字段如下：

表 6-36 pg_rm_residualfiles(filepath)返回字段

名称	类型	描述
result	bool	是否已经完成删除。

示例：

```
SELECT * FROM pg_rm_residualfiles('pgrf_20200908160211441599');
 result
--------
 t
(1 row)
```

说明

- 残留文件只有在调用pg_verify_residualfiles()进行verify后才能被真正删除。
- 删除动作不区分数据库，指定文件中所有已经verify的文件都会被删除。
- 如果指定文件中记录的所有文件都已经被删除，指定文件会被移除并备份到$PGDATA/pg_residualfile/backup目录下。

pg_rm_residualfiles()

描述：用于删除当前实例中所有的残留文件列表中的文件。该函数为实例级函数，与当前所在的数据库无关，可以在任意实例上运行。

参数类型：无

返回值类型：record

函数返回字段如下：

表 6-37 pg_rm_residualfiles()返回字段

名称	类型	描述
result	bool	是否已经完成删除。
filepath	text	残留文件记录路径。
notes	text	注释。

示例：

```
SELECT * FROM pg_rm_residualfiles();
 result |          filepath           | notes
--------+-----------------------------+-------
 t      | pgrf_20200908160211441546 |
(1 row)
```

说明

- 残留文件只有在调用pg_verify_residualfiles()进行验证后才能被真正删除。
- 删除动作不区分数据库，指定文件中所有已经验证的文件都会被删除。
- 如果指定文件中记录的所有文件都已经被删除，指定文件会被移除并备份到$PGDATA/pg_residualfile/backup目录下。

pgxc_rm_residualfiles()

描述：pgxc_rm_residualfiles的CN统一查询函数。该函数为集群级函数，与当前所在的数据库无关，在CN实例上运行。

参数类型：无

返回值类型：record

函数返回字段如下：

表 6-38 pgxc_rm_residualfiles()返回字段

名称	类型	描述
nodename	text	节点名。
result	bool	是否已经完成删除。
filepath	text	残留文件记录路径。

名称	类型	描述
notes	text	注释。

示例：

```
SELECT * FROM pgxc_rm_residualfiles();
   nodename   | result |          filepath          | notes
--------------+--------+----------------------------+-------
 cn_5001      | t      | pgrf_20200910170129360401 |
 dn_6001_6002 | t      | pgrf_20200908160211441546 |
(2 rows)
```

6.29.4 残留文件管理函数应用

使用步骤

步骤1 调用pgxc_get_residualfiles()函数，获取存在残留文件的数据库名称。

步骤2 分别进入确认有残留文件的数据库，调用pgxc_verify_residualfiles()函数，对当前数据库中记录的残留文件进行验证。

步骤3 调用pgxc_rm_residualfiles()函数，删除所有已经验证过的残留文件。

----结束

说明

pgxc类残留文件管理函数只对CN和当前主DN进行操作，不会验证和清理备DN上的残留文件。所以主DN完成清理后，应在备DN上及时执行残留文件清理操作或对备机进行build，防止主备切换后由于增量build导致备机残留文件被重新复制回主DN，导致未成功清理的假象。

使用示例

以当前两个用户自建的数据库db1、db2为例：

```
postgres=# \l
                                 List of databases
   Name    | Owner  | Encoding  | Collate | Ctype | Access privileges
-----------+--------+-----------+---------+-------+-------------------
 db1       | fpi810 | SQL_ASCII | C       | C     |
 db2       | fpi810 | SQL_ASCII | C       | C     |
 postgres  | fpi810 | SQL_ASCII | C       | C     |
 template0 | fpi810 | SQL_ASCII | C       | C     | =c/fpi810        +
           |        |           |         |       | fpi810=CTc/fpi810
 template1 | fpi810 | SQL_ASCII | C       | C     | =c/fpi810        +
           |        |           |         |       | fpi810=CTc/fpi810
(5 rows)
```

1. 在CN上获取集群的所有残留文件记录：

```
db1=# SELECT * FROM pgxc_get_residualfiles() order by 4, 6; -- order by不是必须的
```

```
db1=# select * from pgxc_get_residualfiles() order by 4, 6;
   nodename    | isverified | isdeleted | dbname |   residualfile   |         filepath          | notes
---------------+------------+-----------+--------+------------------+---------------------------+-------
 dn_6001_6002 | f          | f         | db1    | base/16390/16395 | pgrf_20200921153355979438 |
 dn_6001_6002 | f          | f         | db1    | base/16390/24592 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24579 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24582 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24584 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24585 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24588 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24589 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24591 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24576 | pgrf_20200921153612088342 |
(10 rows)
```

当前集群中：

- dn_6001_6002 节点（当前的主节点实例）的db1和db2数据库中都存在残留文件记录。
- 残留文件在residualfile 列展示。
- filepath列为记录残留文件的记录文件，保存在实例数据目录下pg_residualfiles目录中。

2. 调用pgxc_verify_residualfiles() 函数对db1库进行验证：

```
db1=# SELECT * FROM pgxc_verify_residualfiles();
```

```
postgres=# \c db1
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "db1" as user "fpi810".
db1=# select * from pgxc_verify_residualfiles();
   nodename    | result |         filepath          | notes
---------------+--------+---------------------------+-------
 dn_6001_6002 | t      | pgrf_20200921153355979438 |
 dn_6001_6002 | t      | pgrf_20200921153612088342 |
(2 rows)
```

因为**verify类函数都是database级别**，所以当前在db1库中调用verify函数时，只对属于db1的残留文件进行验证。

可以再次调用get函数查看是否验证完成：

```
db1=# SELECT * FROM pgxc_get_residualfiles() order by 4, 6;
```

```
db1=# select * from pgxc_get_residualfiles() order by 4, 6;
   nodename    | isverified | isdeleted | dbname |   residualfile   |         filepath          | notes
---------------+------------+-----------+--------+------------------+---------------------------+-------
 dn_6001_6002 | t          | f         | db1    | base/16390/16395 | pgrf_20200921153355979438 |
 dn_6001_6002 | t          | f         | db1    | base/16390/24592 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24579 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24582 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24584 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24585 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24588 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24589 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24591 | pgrf_20200921153612088342 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24576 | pgrf_20200921153612088342 |
(10 rows)
```

如上图所示，已确认db1数据库中的残留文件都已经验证，db2数据库中的残留文件都未进行验证。

3. 调用 pgxc_rm_residualfiles()函数删除残留文件。

```
db1=# SELECT * FROM pgxc_rm_residualfiles();
```

```
db1=# select * from pgxc_rm_residualfiles();
   nodename    | result |         filepath          | notes
---------------+--------+---------------------------+-------
 dn_6001_6002 | t      | pgrf_20200921153355979438 |
 dn_6001_6002 | t      | pgrf_20200921153612088342 |
(2 rows)
```

4. 再次调用pgxc_get_residualfiles()函数检查删除后的结果。

```
db1=# select * from pgxc_get_residualfiles() order by 4, 6;
   nodename   | isverified | isdeleted | dbname |   residualfile   |         filepath          | notes
--------------+------------+-----------+--------+------------------+---------------------------+-------
 dn_6001_6002 | t          | t         | db1    | base/16390/24592 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24579 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24582 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24584 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24576 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24588 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24589 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24591 | pgrf_2020092115361208834 2 |
 dn_6001_6002 | f          | f         | db2    | base/16391/24585 | pgrf_2020092115361208834 2 |
(9 rows)
```

结果显示db1数据库中的残留文件已经被删除（isdeleted标记为t），db2中的残留文件都未被删除。

同时可以看到查询出9条结果，与之前查询出的结果想比，缺少一条以9438结尾的残留文件记录文件。这是因为以9438结尾的残留文件记录文件中只有一条残留文件记录，这条记录在步骤3中被删除，当记录文件中的所有残留文件都被删除后，记录文件本身也会被删除，并备份到pg_residualfiles/backup目录中：

```
[fpi810@host-192-168-244-162 pg_residualfiles]$ ls
backup  pendingdeletesfile  pgrf_20200921153612088342
[fpi810@host-192-168-244-162 pg_residualfiles]$ ls backup/
pgrf_20200921153355979438_bak
```

5. 如果需要删除db2数据库中的文件，需要在db2中调用verify函数后再调用rm函数。

   a. 进入db2数据库，并调用验证函数：

   ```
   db1=# \c db2
   Non-SSL connection (SSL connection is recommended when requiring high-security)
   You are now connected to database "db2" as user "fpi810".
   db2=# select * from pgxc_verify_residualfiles();
      nodename   | result |         filepath          | notes
   --------------+--------+---------------------------+-------
    dn_6001_6002 | t      | pgrf_20200921153612088342 |
   (1 row)
   ```

   此时可以查询验证的结果：

   ```
   db2=# select * from pgxc_get_residualfiles() order by 4, 6;
      nodename   | isverified | isdeleted | dbname |   residualfile   |         filepath          | notes
   --------------+------------+-----------+--------+------------------+---------------------------+-------
    dn_6001_6002 | t          | t         | db1    | base/16390/24592 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24579 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24582 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24584 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24576 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24588 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24589 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24591 | pgrf_20200921153612088342 |
    dn_6001_6002 | t          | f         | db2    | base/16391/24585 | pgrf_20200921153612088342 |
   (9 rows)
   ```

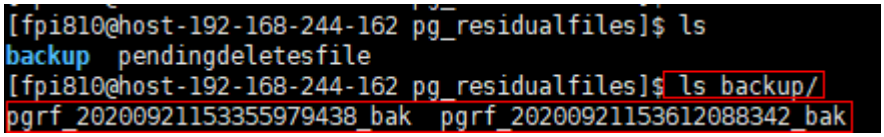
   b. 调用删除函数：

   ```
   db2=# select * from pgxc_rm_residualfiles();
      nodename   | result |         filepath          | notes
   --------------+--------+---------------------------+-------
    dn_6001_6002 | t      | pgrf_20200921153612088342 |
   (1 row)
   ```

   c. 再查询删除的结果：

   ```
   db2=# select * from pgxc_get_residualfiles() order by 4, 6;
    nodename | isverified | isdeleted | dbname | residualfile | filepath | notes
   ----------+------------+-----------+--------+--------------+----------+-------
   (0 rows)
   ```

此时因为 8342 结尾的记录文件中残留文件已经全部删除，所以整个记录文件也被删除并备份到backup目录下，所以查询到0条记录。



6.30 统计信息函数

统计信息函数根据访问对象分为两种类型：

- 针对某个数据库进行访问的函数，以数据库中每个表或索引的OID作为参数，标识需要报告的数据库。
- 针对某个服务器进行访问的函数，以一个服务器进程号为参数，其范围从1到当前活跃服务器的数目。

pg_stat_get_db_numbackends(oid)

描述：查询当前实例上指定数据库活跃的服务器线程数目。

返回值类型：integer

pg_stat_get_db_total_numbackends(oid)

描述：在CN上执行该函数，返回集群中所有CN上指定数据库活跃的服务器线程总数。在DN上执行该函数，返回当前实例上指定数据库活跃的服务器线程数目。

返回值类型：integer

pg_stat_get_db_xact_commit(oid)

描述：返回当前实例上指定数据库中已提交事务的数量。

返回值类型：bigint

pg_stat_get_db_total_xact_commit(oid)

描述：在CN上执行该函数，返回集群中所有CN上指定数据库中已提交事务的总数。在DN上执行该函数，返回当前实例上指定数据库中已提交事务的数量。

返回值类型：bigint

pg_stat_get_db_xact_rollback(oid)

描述：返回当前实例上指定数据库中回滚事务的数量。

返回值类型：bigint

pg_stat_get_db_total_xact_rollback(oid)

描述：在CN上执行该函数，返回集群中所有CN上指定数据库中回滚事务的总数。在DN上执行该函数，返回当前实例上指定数据库中回滚事务的数量。

返回值类型：bigint

pg_stat_get_db_blocks_fetched(oid)

描述：返回当前实例上指定数据库中磁盘块抓取请求的数量。

返回值类型：bigint

pg_stat_get_db_total_blocks_fetched(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库中磁盘块抓取请求的总数。在DN上执行该函数，返回当前实例上指定数据库中磁盘块抓取请求的数量。

返回值类型：bigint

pg_stat_get_db_blocks_hit(oid)

描述：返回当前实例上指定数据库在缓冲区中找到的请求磁盘块的数量。

返回值类型：bigint

pg_stat_get_db_total_blocks_hit(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库在缓冲区中找到的请求磁盘块的总数。在DN上执行该函数，返回当前实例上指定数据库在缓冲区中找到的请求磁盘块的数量。

返回值类型：bigint

pg_stat_get_db_tuples_returned(oid)

描述：返回当前实例上指定数据库返回的元祖数量。

返回值类型：bigint

pg_stat_get_db_total_tuples_returned(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库返回的元组总数。在DN上执行该函数，返回当前实例上指定数据库返回的元组数量。

返回值类型：bigint

pg_stat_get_db_tuples_fetched(oid)

描述：返回当前实例上指定数据库中读取的元组数量。

返回值类型：bigint

pg_stat_get_db_total_tuples_fetched(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库读取的元组总数。在DN上执行该函数，返回当前实例上指定数据库读取的元组数量。

返回值类型：bigint

pg_stat_get_db_tuples_inserted(oid)

描述：返回当前实例上指定数据库中插入的元组数量。

返回值类型：bigint

pg_stat_get_db_total_tuples_inserted(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库插入的元组总数。在DN上执行该函数，返回当前实例上指定数据库插入的元组数量。

返回值类型：bigint

pg_stat_get_db_tuples_updated(oid)

描述：返回当前实例上指定数据库中更新的元组数量。

返回值类型：bigint

pg_stat_get_db_total_tuples_updated(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库更新的元组总数。在DN上执行该函数，返回当前实例上指定数据库更新的元组数量。

返回值类型：bigint

pg_stat_get_db_tuples_deleted(oid)

描述：返回当前实例上指定数据库中删除的元组数量。

返回值类型：bigint

pg_stat_get_db_total_tuples_deleted(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库删除的元组总数。在DN上执行该函数，返回当前实例上指定数据库删除的元组数量。

返回值类型：bigint

pg_stat_get_db_conflict_lock(oid)

描述：在CN上执行该函数，返回集群中所有CN和DN上指定数据库锁冲突的总数。在DN上执行该函数，返回当前实例上指定数据库中锁冲突数量。

返回值类型：bigint

pg_stat_get_db_deadlocks(oid)

描述：返回当前实例上指定数据库中死锁的数量。

返回值类型：bigint

pg_stat_get_db_total_deadlocks(oid)

描述：在CN上执行该函数，返回集群中所有CN和DN上指定数据库死锁的总数。在DN上执行该函数，返回当前实例上指定数据库中死锁的数量。

返回值类型：bigint

pg_stat_get_db_conflict_all(oid)

描述：返回当前实例上指定数据库中发生冲突恢复的次数。

返回值类型：bigint

pg_stat_get_db_total_conflict_all(oid)

描述：在CN上执行该函数，返回集群中所有CN和DN上指定数据库发生冲突恢复的总次数。在DN上执行该函数，返回当前实例上指定数据库中发生冲突恢复的次数。

返回值类型：bigint

pg_stat_get_db_temp_files(oid)

描述：返回当前实例上指定数据库中创建临时文件的个数。

返回值类型：bigint

pg_stat_get_db_total_temp_files(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库中创建临时文件的总个数。在DN上执行该函数，返回当前实例上指定数据库中创建临时文件的个数。

返回值类型：bigint

pg_stat_get_db_temp_bytes(oid)

描述：返回当前实例上指定数据库中创建临时文件的字节数。

返回值类型：bigint

pg_stat_get_db_total_temp_bytes(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库中创建临时文件的总字节数。在DN上执行该函数，返回当前实例上指定数据库中创建临时文件的字节数。

返回值类型：bigint

pg_stat_get_db_blk_read_time(oid)

描述：返回当前实例上指定数据库中读数据块所用的时间。

返回值类型：double

pg_stat_get_db_total_blk_read_time(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库中读数据块所用的总时间。在DN上执行该函数，返回当前实例上指定数据库中读数据块所用的时间。

返回值类型：double

pg_stat_get_db_blk_write_time(oid)

描述：返回当前实例上指定数据库中写数据块所用的时间。

返回值类型：double

pg_stat_get_db_total_blk_write_time(oid)

描述：在CN上执行该函数，返回集群中所有DN上指定数据库中写数据块所用的总时间。在DN上执行该函数，返回当前实例上指定数据库中写数据块所用的时间。

返回值类型：double

pg_stat_get_numscans(oid)

描述：如果参数是一个表，则顺序扫描读取的行数目。

如果参数是一个索引，则返回索引行的数目。

返回值类型：bigint

pg_stat_get_tuple()

描述：在CN和DN上均可以执行，该函数仅8.1.3及以上集群版本支持。

函数无参时，查询CN上所有系统表的统计信息及表在每个CN上的脏页率，查询DN上所有系统表和用户表的统计信息和表在每个DN上的脏页率；

函数带入参时，入参是schema和表名，带入参的函数执行时查询单张表的统计信息和脏页率。

说明

该函数的统计信息依赖于ANALYZE，为获取最准确的信息请先对表进行ANALYZE。

返回值类型：record

函数返回字段如下：

表 6-39 pg_stat_get_tuple()返回字段

名称	类型	描述
nodename	text	节点名。
tableid	oid	表的oid。
partid	oid	分区表的分区oid。
last_vacuum	timestamp with time zone	最后一次手动vacuum时间。
last_autovacuum	timestamp with time zone	最后一次autovacuum时间。
last_analyze	timestamp with time zone	最后一次手动analyze时间。
last_autoanalyze	timestamp with time zone	最后一次autoanalyze时间。
vacuum_count	bigint	vacuum次数。

名称	类型	描述
autovacuum_count	bigint	autovacuum次数。
analyze_count	bigint	analyze次数。
autoanalyze_count	bigint	autoanalyze_count次数。
n_tup_ins	bigint	插入的行数。
n_tup_upd	bigint	更新的行数。
n_tup_del	bigint	删除的行数。
n_tup_hot_upd	bigint	HOT更新的行数。
n_tup_change	bigint	analyze之后改变的行数。
n_live_tup	bigint	live行估计数。
n_dead_tup	bigint	dead行估计数。
dirty_rate	bigint	单节点的脏页率（单CN或单DN级）。
last_data_changed	timestamp with time zone	记录表最后一次数据变化的时间。

pg_stat_get_tuples_returned(oid)

描述：如果参数是一个表，则顺序扫描读取的行数目。

如果参数是一个索引，则返回的索引行的数目。

返回值类型：bigint

pg_stat_get_tuples_fetched(oid)

描述：如果参数是一个表，则位图扫描抓取的行数目。

如果参数是一个索引，则用简单索引扫描抓取的行数目。

返回值类型：bigint

pg_stat_get_tuples_inserted(oid)

描述：插入表中行的数量。

返回值类型：bigint

pg_stat_get_local_tuples_inserted(oid)

描述：当前节点上插入表中行的数量，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_tuples_updated(oid)

描述：在表中已更新行的数量。

返回值类型：bigint

pg_stat_get_local_tuples_updated(oid)

描述：当前节点上在表中已更新行的数量，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_tuples_deleted(oid)

描述：从表中删除行的数量。

返回值类型：bigint

pg_stat_get_local_tuples_deleted(oid)

描述：当前节点上从表中删除行的数量，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_tuples_changed(oid)

描述：在CN上查询该函数，返回该表自上一次analyze或autoanalyze之后插入、更新、删除行的总数量。在DN上查询该函数，返回该表在当前节点上自上一次analyze或autoanalyze之后插入、更新、删除行的总数量。

返回值类型：bigint

pg_stat_get_local_tuples_changed(oid)

描述：该表在当前节点上自从上一次analyze或autoanalyze之后插入、更新、删除行的数量。

返回值类型：bigint

pg_stat_get_tuples_hot_updated(oid)

描述：热更新的行数表。

返回值类型：bigint

pg_stat_get_local_tuples_hot_updated(oid)

描述：当前节点上热更新的行数表，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_live_tuples(oid)

描述：表格的活元组数。

返回值类型：bigint

pg_stat_get_local_live_tuples(oid)

描述：当前节点上表格的活元组数，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_dead_tuples(oid)

描述：表格的死元组数。

返回值类型：bigint

pg_stat_get_local_dead_tuples(oid)

描述：当前节点上表格的死元组数，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_blocks_fetched(oid)

描述：表或者索引的磁盘块抓取请求的数量。

返回值类型：bigint

pg_stat_get_blocks_hit(oid)

描述：在缓冲区中找到的表或者索引的磁盘块请求数目。

返回值类型：bigint

pg_stat_get_partition_tuples_inserted(oid)

描述：插入相应表分区中行的数量。

返回值类型：bigint

pg_stat_get_partition_tuples_updated(oid)

描述：在相应表分区中已更新行的数量。

返回值类型：bigint

pg_stat_get_partition_tuples_deleted(oid)

描述：从相应表分区中删除行的数量。

返回值类型：bigint

pg_stat_get_partition_tuples_changed(oid)

描述：该表分区上一次analyze或autoanalyze之后插入、更新、删除行的总数量。

返回值类型：bigint

pg_stat_get_partition_live_tuples(oid)

描述：活行数表分区。

返回值类型：bigint

pg_stat_get_partition_dead_tuples(oid)

描述：死行数表分区。

返回值类型：bigint

pg_stat_get_xact_tuples_inserted(oid)

描述：表相关的活跃子事务中插入的tuple数。

返回值类型：bigint

pg_stat_get_xact_tuples_deleted(oid)

描述：表相关的活跃子事务中删除的tuple数。

返回值类型：bigint

pg_stat_get_xact_tuples_hot_updated(oid)

描述：表相关的活跃子事务中热更新的tuple数。

返回值类型：bigint

pg_stat_get_xact_tuples_updated(oid)

描述：表相关的活跃子事务中更新的tuple数。

返回值类型：bigint

pg_stat_get_xact_partition_tuples_inserted(oid)

描述：表分区相关的活跃子事务中插入的tuple数。

返回值类型：bigint

pg_stat_get_xact_partition_tuples_deleted(oid)

描述：表分区相关的活跃子事务中删除的tuple数。

返回值类型：bigint

pg_stat_get_xact_partition_tuples_hot_updated(oid)

描述：表分区相关的活跃子事务中热更新的tuple数。

返回值类型：bigint

pg_stat_get_xact_partition_tuples_updated(oid)

描述：表分区相关的活跃子事务中更新的tuple数。

返回值类型：bigint

pg_stat_get_last_vacuum_time(oid)

描述：用户在该表上最后一次手动启动清理或者autovacuum线程启动清理的时间。

返回值类型：timestamptz

pg_stat_get_last_autovacuum_time(oid)

描述：autovacuum守护线程在该表上最后一次启动清理的时间。

返回值类型：timestamptz

pg_stat_get_local_last_autovacuum_time(oid)

描述：当前节点的autovacuum守护线程在该表上最后一次启动清理的时间，该函数仅8.1.2及以上版本支持。

返回值类型：timestamptz

pg_stat_get_vacuum_count(oid)

描述：用户在该表上手动启动清理的次数。

返回值类型：bigint

pg_stat_get_autovacuum_count(oid)

描述：autovacuum守护线程在该表上启动清理的次数。

返回值类型：bigint

pg_stat_get_local_autovacuum_count(oid)

描述：当前节点上的autovacuum守护线程在该表上启动清理的次数，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_last_analyze_time(oid)

描述：用户在该表上最后一次手动启动分析或者autovacuum线程启动分析的时间。

返回值类型：timestamptz

pg_stat_get_last_autoanalyze_time(oid)

描述：autovacuum守护线程在该表上最后一次启动分析的时间。

返回值类型：timestamptz

pg_stat_get_local_last_autoanalyze_time(oid)

描述：当前节点的autovacuum守护线程在该表上最后一次启动分析的时间，该函数仅8.1.2及以上版本支持。

返回值类型：timestamptz

pg_stat_get_analyze_count(oid)

描述：用户在该表上手动启动分析的次数。

返回值类型：bigint

pg_stat_get_autoanalyze_count(oid)

描述：autovacuum守护线程在该表上启动分析的次数。

返回值类型：bigint

pg_stat_get_local_autoanalyze_count(oid)

描述：当前节点的autovacuum守护线程在该表上启动分析的次数，该函数仅8.1.2及以上版本支持。

返回值类型：bigint

pg_stat_get_local_analyze_status(oid)

描述：指定表在当前节点上的是否需要analyze的状态，仅在CN端有意义。该函数仅8.1.2及以上版本支持。

- 如果该表的修改行数超过analyze的阈值（根据autovacuum_analyze_threshold + autovacuum_analyze_scale_factor * reltuples计算，其中reltuples是pg_class中记录的表的估算行数），则返回“Analyze needed”。
- 如果该表的修改行数不超过analyze的阈值，则返回“Analyze not needed”。
- 如果该表正在进行由查询触发的analyze，则返回“Analyze in progress”。
- 如果该表是否需要analyze的状态未知，则返回“Unknown analyze status”。

返回值类型：text

pg_total_autovac_tuples(bool)

描述：返回total autovac相关的tuple记录，如nodename,nspname,relname以及各类tuple的IUD信息。

返回值类型：setof record

pg_autovac_status(oid)

描述：返回和autovac状态相关的参数信息，如nodename,nspname,relname,analyze,vacuum设置，analyze/vacuum阈值, analyze/vacuum tuple数等。

返回值类型：setof record

pg_autovac_timeout(oid)

描述：返回某个表做autovac连续超时的次数，表信息非法或node信息异常返回NULL。

返回值类型：bigint

pg_autovac_coordinator(oid)

描述：返回对某个表做autovac的coordinator名字，表信息非法或node信息异常返回NULL。

返回值类型：text

pgxc_get_wlm_session_info_bytime(text, timestamp without time zone, timestamp without time zone, int)

描述：PGXC_WLM_SESSION_INFO视图在统计数据量很大的场景中性能较差，建议使用该函数进行筛选查询。入参分别为：筛选时间列（ 'start_time', 'finish_time' ），起始区间时间，结束区间时间，每个CN返回的最大数量。返回值为GS_WLM_SESSION_HISTORY。

返回值类型：setof record

pgxc_get_wlm_current_instance_info(text, int default null)

描述：在CN节点上查询集群各节点当前的资源使用情况，读取内存中还未存到GS_WLM_INSTANCE_HISTORY系统表的数据。入参分别为：节点名称(可以输入ALL、C、D、实例名称)，每个节点返回的最大数量。返回值为GS_WLM_INSTANCE_HISTORY。

返回值类型：setof record

pgxc_get_wlm_history_instance_info(text, TIMESTAMP, TIMESTAMP, int default null)

描述：在CN节点上查询集群各节点历史资源使用情况，读取GS_WLM_INSTANCE_HISTORY系统表的数据。入参分别为：节点名称(可以输入ALL、C、D、实例名称)，起始区间时间，结束区间时间，每个实例返回的最大数量。返回值为GS_WLM_INSTANCE_HISTORY。

返回值类型：setof record

pg_stat_get_last_data_changed_time(oid)

描述：insert/update/delete, exchange/drop partition在该表上最后一次操作的时间，PG_STAT_ALL_TABLES视图last_data_changed列的数据是通过该函数求值，在表数量很大的场景中，通过视图获取表数据最后修改时间的性能较差，建议直接使用该函数获取表数据的最后修改时间。

返回值类型：timestamptz

pg_stat_set_last_data_changed_time(oid)

描述：手动设置该表上最后一次insert/update/delete, exchange/truncate/drop partition操作的时间。

返回值类型：void

pv_session_time()

描述：统计当前节点各会话线程的运行时间信息及各执行阶段所消耗时间。

返回值类型：record

pv_instance_time()

描述：统计当前节点的运行时间信息及各执行阶段所消耗时间。

返回值类型：record

pg_stat_get_activity(integer)

描述：返回一个关于带有特殊PID的后台进程的记录信息，当参数为NULL时，则返回每个活动的后台进程的记录。返回结果是PG_STAT_ACTIVITY视图中的一个子集，不包含connection_info列。

返回值类型：setof record

pg_stat_get_activity_with_conninfo(integer)

描述：返回一个关于带有特殊PID的后台进程的记录信息，当参数为NULL时，则返回每个活动的后台进程的记录。返回结果是PG_STAT_ACTIVITY视图中的一个子集。

返回值类型：setof record

pg_user_iostat(text)

描述：该函数8.1.2版本中已废弃，为兼容历史版本功能保留该函数，当前版本查询无效。

返回值类型：record

表 6-40 pg_user_iostat(text)返回字段

名称	类型	描述
userid	oid	用户ID。
min_curr_iops	int4	当前该用户IO在各DN中的最小值。
max_curr_iops	int4	当前该用户IO在各DN中的最大值。
min_peak_iops	int4	该用户IO峰值中，各DN的最小值。
max_peak_iops	int4	该用户IO峰值中，各DN的最大值。
io_limits	int4	用户指定的资源池所设置的io_limits。
io_priority	text	该用户所设io_priority。

pg_stat_get_function_calls(oid)

描述：函数已被调用次数。

返回值类型：bigint

pg_stat_get_function_total_time(oid)

描述：该函数花费的总挂钟时间，以微秒为单位。包括花费在此函数调用上的时间。

返回值类型：double precision

pg_stat_get_function_self_time(oid)

描述：在当前事务中仅花费在此函数上的时间。不包括花费在调用函数上的时间。

返回值类型：double precision

pg_stat_get_backend_idset()

描述：设置当前活动的服务器进程数（从1到活动服务器进程的数量）。

返回值类型：setofinteger

pg_stat_get_backend_pid(integer)

描述：给定的服务器线程的线程ID。

返回值类型：bigint

```
SELECT pg_stat_get_backend_pid(1);
 pg_stat_get_backend_pid
-------------------------
        139706243217168
(1 row)
```

pg_stat_get_backend_dbid(integer)

描述：连接到给定服务器进程的数据库ID。

返回值类型：oid

pg_stat_get_backend_userid(integer)

描述：给定的服务器进程的用户ID。

返回值类型：oid

pg_stat_get_backend_activity(integer)

描述：给定服务器进程的当前活动查询，仅在调用者是系统管理员或被查询会话的用户，并且打开track_activities的时候才能获得结果。

返回值类型：text

pg_stat_get_backend_waiting(integer)

描述：如果给定服务器进程在等待某个锁，并且调用者是系统管理员或被查询会话的用户，并且打开track_activities的时候才返回真。

返回值类型：boolean

pg_stat_get_backend_activity_start(integer)

描述：给定服务器进程当前正在执行的查询的起始时间，仅在调用者是系统管理员或被查询会话的用户，并且打开track_activities的时候才能获得结果。

返回值类型：timestampwithtimezone

pg_stat_get_backend_xact_start(integer)

描述：给定服务器进程当前正在执行的事务的开始时间，但只有当前用户是系统管理员或被查询会话的用户，并且打开track_activities的时候才能获得结果。

返回值类型：timestampwithtimezone

pg_stat_get_backend_start(integer)

描述：给定服务器进程启动的时间，如果当前用户不是系统管理员或被查询的后端的用户，则返回NULL。

返回值类型：timestampwithtimezone

pg_stat_get_backend_client_addr(integer)

描述：连接到给定客户端后端的IP地址。

如果是通过Unix域套接字连接的则返回NULL；如果当前用户不是系统管理员或被查询会话的用户，也返回NULL。

返回值类型：inet

备注：该函数中IP地址作为入参时，需写为不带圆点的格式，例如，192.168.100.128写为192168100128。

pg_stat_get_backend_client_port(integer)

描述：连接到给定客户端后端的TCP端口。

如果是通过Unix域套接字连接的则返回-1；如果当前用户不是系统管理员或被查询会话的用户，也返回NULL。

返回值类型：integer

pg_stat_get_bgwriter_timed_checkpoints()

描述：后台写进程开启定时检查点的时间（因为checkpoint_timeout时间已经过期了）。

返回值类型：bigint

pg_stat_get_bgwriter_requested_checkpoints()

描述：后台写进程开启基于后端请求的检查点的时间，因为已经超过了checkpoint_segments或因为已经执行了CHECKPOINT。

返回值类型：bigint

pg_stat_get_bgwriter_buf_written_checkpoints()

描述：在检查点期间后台写进程写入的缓冲区数目。

返回值类型：bigint

pg_stat_get_bgwriter_buf_written_clean()

描述：为日常清理脏块，后台写进程写入的缓冲区数目。

返回值类型：bigint

pg_stat_get_bgwriter_maxwritten_clean()

描述：后台写进程停止清理扫描的时间，因为已经写入了更多的缓冲区（相比bgwriter_lru_maxpages参数声明的缓冲区数）。

*返回值类型：b*igint

pg_stat_get_buf_written_backend()

描述：后端进程写入的缓冲区数，因为它们需要分配一个新的缓冲区。

返回值类型：bigint

pg_stat_get_buf_alloc()

描述：分配的总缓冲区数。

返回值类型：bigint

pg_stat_clear_snapshot()

描述：清理当前的统计快照。

返回值类型：void

pg_stat_reset()

描述：为当前数据库重置统计计数器为0（需要系统管理员权限）。

返回值类型：void

pg_stat_reset_shared(text)

描述：重置shared cluster每个节点当前数据统计计数器为0（需要系统管理员权限）。

返回值类型：void

pg_stat_reset_single_table_counters(oid)

描述：为当前数据库中的一个表或索引重置统计为0（需要系统管理员权限）。

返回值类型：void

pg_stat_reset_single_function_counters(oid)

描述：为当前数据库中的一个函数重置统计为0（需要系统管理员权限）。

返回值类型：void

pg_stat_session_cu(int, int, int)

描述：获取当前节点所运行session的CU命中统计信息。

返回值类型：record

gs_get_stat_session_cu(text, int, int, int)

描述：获取集群所有运行session的CU命中统计信息。

返回值类型：record

gs_get_stat_db_cu(text, text, bigint, bigint, bigint)

描述：获取集群一个数据库的CU命中统计信息。

返回值类型：record

pg_stat_get_cu_mem_hit(oid)

描述：获取当前节点当前数据库中一个列存表的CU内存命中次数。

返回值类型：bigint

pg_stat_get_cu_hdd_sync(oid)

描述：获取当前节点当前数据库中一个列存表从磁盘同步读取CU次数。

返回值类型：bigint

pg_stat_get_cu_hdd_asyn(oid)

描述：获取当前节点当前数据库中一个列存表从磁盘异步读取CU次数。

返回值类型：bigint

pg_stat_get_db_cu_mem_hit(oid)

描述：获取当前节点一个数据库CU内存命中次数。

返回值类型：bigint

pg_stat_get_db_cu_hdd_sync(oid)

描述：获取当前节点一个数据库从磁盘同步读取CU次数。

返回值类型：bigint

pg_stat_get_db_cu_hdd_asyn(oid)

描述：获取当前节点一个数据库从磁盘异步读取CU次数。

返回值类型：bigint

pgxc_fenced_udf_process()

描述：查看UDF Master和Work进程数。

返回值类型：record

pgxc_terminate_all_fenced_udf_process()

描述：Kill所有的UDF Work进程。

返回值类型：bool

GS_ALL_NODEGROUP_CONTROL_GROUP_INFO(text)

描述：提供了所有逻辑集群的控制组信息。该函数在调用的时候需要指定要查询逻辑集群的名称。例如要查询'installation'逻辑集群的控制组信息：

```
SELECT * FROM GS_ALL_NODEGROUP_CONTROL_GROUP_INFO('installation')
```

返回值类型：record

函数返回字段如下：

表 6-41 GS_ALL_NODEGROUP_CONTROL_GROUP_INFO(text)

名称	类型	描述
name	text	控制组的名称。
type	text	控制组的类型。
gid	bigint	控制组ID。
classgid	bigint	Workload所属Class的控制组ID。
class	text	Class控制组。
workload	text	Workload控制组。
shares	bigint	控制组分配的CPU资源配额。
limits	bigint	控制组分配的CPU资源限额。
wdlevel	bigint	Workload控制组层级。
cpucores	text	控制组使用的CPU核的信息。

gs_get_nodegroup_tablecount(name)

描述：得到一个逻辑集群中所有数据库包含的用户表数目。

返回值类型：integer

pgxc_max_datanode_size(name)

描述：得到一个逻辑集群的所有DN节点中数据库文件占用磁盘空间的最大值，单位为字节。

返回值类型：bigint

gs_check_logic_cluster_consistency()

描述：检查当前系统中所有逻辑集群是否存在系统信息不一致的情况，如果返回空记录，表示不存在不一致情况；否则，逻辑集群中CN和DN上的NodeGroup信息存在不一致。该函数应该在非扩缩容重分布时调用。

返回值类型：record

gs_check_tables_distribution()

描述： 检查当前系统中用户表的分布是否存在不一致，如果返回空记录，表示不存在不一致。该函数应该在非扩缩容重分布时调用。

返回值类型：record

pg_stat_bad_block(text, int, int, int, int, int, timestamp with time zone, timestamp with time zone)

描述：获取当前节点自启动后，读取出现Page/CU的损坏信息。

返回值类型：record

pgxc_stat_bad_block(text, int, int, int, int, int, timestamp with time zone, timestamp with time zone)

描述：获取集群所有节点自启动后，读取出现Page/CU的损坏信息。

返回值类型：record

pg_stat_bad_block_clear()

描述：清理节点记录的读取出现的Page/CU损坏信息（需要系统管理员权限）。

返回值类型：void

pgxc_stat_bad_block_clear()

描述：清理集群所有节点记录的读取出现的Page/CU损坏信息（需要系统管理员权限）。

返回值类型：void

gs_respool_exception_info(pool text)

描述：查看某个资源池关联的查询规则信息。

返回值类型：record

gs_control_group_info(pool text)

描述：查看资源池关联的控制组信息

返回值类型：record

返回信息如下：

表 6-42 gs_control_group_info(pool text)返回字段

属性	属性值	描述
name	class_a:workload_a1	Class和workload名称。
class	class_a	Class控制组名称。
workload	workload_a1	Workload控制组名称。
type	DEFWD	控制组类型（Top、CLASS、BAKWD、DEFWD、TSWD）。
gid	87	控制组ID。
shares	30	占父节点CPU资源的百分比。
limits	0	占父节点CPU核数的百分比。
rate	0	Timeshare中的分配比例。
cpucores	0-3	CPU核心数。

gs_wlm_user_resource_info(name text)

描述：查询具体某个用户的资源限额和资源使用情况。

返回值类型：record

pgxc_stat_single_table(schema, talename)

描述：在CN上执行，入参为schema和表名。查询单张表在全库中的统计信息及该表在每个DN上的脏页率。

该函数仅8.1.3及以上集群版本支持。

说明

该函数的统计信息依赖于ANALYZE，为获取该表最准确的信息请先对表进行ANALYZE。

返回值类型：record

返回值字段与函数**pg_stat_get_tuple()**相同。

```
SELECT * FROM pgxc_stat_single_table('public','t1');
 nodename  | tableid | partid |      last_vacuum       |    last_autovacuum     |        last_analyze         |
last_autoanalyze     | vacuum_count | autovacuum_count | analyze_count | autoanalyze_count | n_tup_ins | n_
tup_upd | n_tup_del | n_tup_hot_upd | n_tup_change | n_live_tup | n_dead_tup | dirty_rate |
last_data_changed
-----------+---------+--------+------------------------+------------------------+-----------------------------
+------------------------+--------------+------------------+---------------+-------------------+-----------+---
```

```
--------+-----------+---------------+--------------+------------+------------+------------+-------------------
 datanode1 | 1270075 |        | 2000-01-01 08:00:00+08 | 2000-01-01 08:00:00+08 | 2023-01-09
09:38:43.220876+08 | 2000-01-01 08:00:00+08 |             0 |               0 |             1 |               0 |         0 |
       0 |         0 |             0 |             0 |          0 |          0 |          0 |
(1 row)
```

6.31 触发器函数

pg_get_triggerdef(oid)

描述：获取触发器的定义信息。

参数：待查触发器的OID。

返回值类型：text

示例：

```
SELECT pg_get_triggerdef(oid) FROM pg_trigger;
                                                pg_get_triggerdef
-----------------------------------------------------------------------------------------------------------------------------
 CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl FOR EACH ROW EXECUTE
PROCEDURE tri_insert_func()
(1 row)
```

pg_get_triggerdef(oid, boolean)

描述：获取触发器的定义信息。

参数：待查触发器的OID及是否以pretty方式展示。

返回值类型：text

说明

仅在创建trigger时指定WHEN条件的情况下，布尔类型参数才生效。

示例：

```
SELECT pg_get_triggerdef(oid,true) FROM pg_trigger;
                                                pg_get_triggerdef
-----------------------------------------------------------------------------------------------------------------------------
 CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl FOR EACH ROW EXECUTE
PROCEDURE tri_insert_func()
(1 row)

SELECT pg_get_triggerdef(oid,false) FROM pg_trigger;
                                                pg_get_triggerdef
-----------------------------------------------------------------------------------------------------------------------------
 CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl FOR EACH ROW EXECUTE
PROCEDURE tri_insert_func()
(1 row)
```

6.32 XML 函数

6.32.1 产生 XML 内容

本节函数和类函数的表达式可以用来从SQL数据产生XML内容。适用于将查询结果格式化成XML文档以便于在客户端应用中处理。

XMLPARSE ({ DOCUMENT | CONTENT } *value*)

描述：从字符数据中生成一个XML类型的值。

返回值类型：xml

示例：

```
SELECT xmlparse(document '<foo>bar</foo>');
   xmlparse
----------------
 <foo>bar</foo>
(1 row)
```

XMLSERIALIZE ({ DOCUMENT | CONTENT } *value* AS *type*

描述：从XML类型的值生成一个字符串。

返回值类型：type，可以是character，character varying或text（或其别名）

示例：

```
SELECT xmlserialize(content 'good' AS CHAR(10));
 xmlserialize
--------------
 good
(1 row)
```

xmlcomment(text)

描述：用于创建一个XML值，它含有一个以指定文本作为内容的XML注释。该文本中不能包含“--”或以一个“-”结尾，这样的文本才是有效的XML注释。当参数为空时，结果也为空。

返回值类型：xml

示例：

```
SELECT xmlcomment('hello');
  xmlcomment
--------------
 <!--hello-->
(1 row)
```

xmlconcat(xml[, ...])

描述：用于将XML值组成的列表串接成一个单独的值。其中空值会被忽略，当所有参数都为空时，结果都为空。

返回值类型：xml

示例：

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');
     xmlconcat
----------------------
 <abc/><bar>foo</bar>
(1 row)
```

说明：XML声明如果存在，结果如下：如果所有参数值都使用相同的XML版本声明，则在结果中使用版本，否则不用版本。如果所有参数值都有standalone属性，且值都为yes， 则结果中standalone值为yes，如果至少有一个是no，则结果中standalone值为no。否则结果中将不带standalone属性。

示例：

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');
           xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
(1 row)
```

xmlelement(name name [, xmlattributes(value [AS attname] [, ...])] [, content, ...])

描述：使用给定名称、属性和内容产生一个XML元素。

返回值类型：xml

示例：

```
SELECT xmlelement(name foo);
 xmlelement
------------
 <foo/>
(1 row)

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));
   xmlelement
------------------
 <foo bar="xyz"/>
(1 row)

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');
            xmlelement
---------------------------------------
 <foo bar="2023-08-16">content</foo>
(1 row)
```

没有合法XML名称的元素和属性名会被转义，转义的方式是将不合法的字符用序列_xHHHH_替换，其中HHHH是该字符以16进制表达的Unicode代码点。例如：

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

           xmlelement
---------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

如果属性值是一个列引用，则不需要指定显式的属性名，这种情况下默认将把列名用作该属性的名字。在其他情况下，必须给该属性一个显式的命名。因此这个例子是合法的：

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

但这些就不合法：

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

元素内容（如果指定）将被根据其数据类型进行格式化。如果内容本身是类型xml，则会构建复杂的XML文档。例如：

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),xmlelement(name
abc),xmlcomment('test'),xmlelement(name xyz));
                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```


其他类型的内容将被格式化为合法的XML字符数据。这意味着特别的字符<、>以及&将会被转换成实体。二进制数据（数据类型bytea）将被表示为base64或者十六进制编码，具体取决于配置参数xmlbinary。

xmlforest(content [AS name] [, ...])

描述：使用给定名称和内容产生一个元素的XML森林（序列）。

返回值类型：xml

示例：

```
SELECT xmlforest('abc' AS foo, 123 AS bar);
          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>
(1 row)

SELECT xmlforest(table_name, column_name) FROM ALL_TAB_COLUMNS WHERE schema = 'pg_catalog';
                                                xmlforest
-----------------------------------------------------------------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 <table_name>pg_authid</table_name><column_name>rolinherit</column_name>
 <table_name>pg_authid</table_name><column_name>rolcreaterole</column_name>
```

xmlpi(name target [, content])

描述：创建一个XML处理指令。内容不能包含字符序列?>

返回值类型：xml

示例：

```
SELECT xmlpi(name php, 'echo "hello world";');
            xmlpi
-----------------------------
 <?php echo "hello world";?>
(1 row)
```

xmlroot(xml, version text | no value [, standalone yes|no|no value])

描述：修改一个XML值的根节点的属性。如果指定了一个版本，它会替换根节点的版本声明中的值；如果指定了一个standalone值，它会替换根节点standalone中的值。

返回值类型：xml

示例：

```
SELECT xmlroot(xmlparse(document '<?xml version="1.0" standalone="no"?><content>abc</content>'),
version '1.1', standalone yes);
                xmlroot
----------------------------------------------------------------
 <?xml version="1.1" standalone="yes"?><content>abc</content>
(1 row)
```

xmlagg(xml)

描述：函数xmlagg是一个聚集函数，将输入值串接起来。

返回值类型：xml

示例：


```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');

SELECT xmlagg(x) FROM test;
       xmlagg
----------------------
 <bar/><foo>abc</foo>
(1 row)
```

为聚集调用增加一个ORDER BY子句即可决定串接的顺序，例如：

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
       xmlagg
----------------------
 <bar/><foo>abc</foo>
(1 row)
```

6.32.2 XML 谓词

本节的函数用户检查xml值的属性。

xml IS DOCUMENT

描述：如果参数XML值是一个正确的XML文档，则IS DOCUMENT返回真；如果非正确XML文档，则返回假；参数为空时返回空。

返回值类型：bool

```
SELECT '<abc/>' is document;
 ?column?
----------
 t
(1 row)
```

xml IS NOT DOCUMENT

描述：如果参数XML值不是一个正确的XML文档，则IS NOT DOCUMENT返回真；如果是正确XML文档，则返回假；参数为空时返回空。

返回值类型：bool

```
SELECT 'abc' is document;
 ?column?
----------
 f
(1 row)
```

XMLEXISTS(text PASSING [BY REF] xml [BY REF])

描述：如果第一个参数中的XPath表达式返回任何节点，则函数XMLEXISTS返回真，否则返回假（如果哪一个参数为空，则结果就为空）。BY REF子句没有作用，用于保持SQL兼容性。

返回值类型：bool

示例：

```
SELECT xmlexists('//town[text() = ''TScity'']' PASSING BY REF '<towns><town>TScity</town><town>TOcity</
town></towns>');
 xmlexists
------------
 t
(1 row)
```

xml_is_well_formed(text)

描述：检查text字符串是不是格式良好的XML值，返回布尔结果。如果xmloption参数设置为DOCUMENT则检查文档，如果设置为CONTENT则检查内容。

返回值类型：bool

示例：

```
SET xmloption TO DOCUMENT;
SET

SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)
```

xml_is_well_formed_document(text)

描述：检查text字符串是不是格式良好的文档，返回布尔结果。

返回值类型：bool

示例：

```
SELECT xml_is_well_formed_document('<test:foo xmlns:test="http://test.com/test">bar</test:foo>');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<test:foo xmlns:test="http://test.com/test">bar</my:foo>');
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

xml_is_well_formed_content(text)

描述：检查text字符串是不是格式良好的内容，返回布尔结果。

返回值类型：bool

示例：

```
SELECT xml_is_well_formed_content('content');
 xml_is_well_formed_content
----------------------------
 t
(1 row)

SELECT xml_is_well_formed_content('<content');
```

```
 xml_is_well_formed_content
----------------------------
 f
(1 row)
```

6.32.3 处理 XML

为了处理数据类型XML的值，GaussDB(DWS)提供了函数xpath和xpath_exists计算XPath表达式以及XMLTABLE表函数。

xpath(xpath, xml [, nsarray])

描述：它返回一个XML值的数组对应xpath表达式所产生的节点集。如果xpath表达式返回一个标量值而不是节点集，将返回一个单个元素的数组。

该函数的第二个参数xml必须是一个完整的XML文档，它必须有一个根节点元素。

第三个参数为可选参数，是一个命名空间的数组映射。这个数组应该是一个二维text数组，第二个维的长度等于2（它应该是一个数组的数组，其中每一个正好包含2个元素）。每个数组项的第一个元素是命名空间名称的别名，第二个元素是命名空间URI。这个数组中提供的别名不必与XML文档本身中使用的别名相同。换句话说，在XML文档和xpath函数上下文中，别名都是本地的。

返回值类型：xml值的数组

示例：

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>', ARRAY[ARRAY['my',
'http://example.com']]);
 xpath
--------
 {test}
(1 row)
```

要处理默认（匿名）命名空间：

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></
a>',ARRAY[ARRAY['mydefns', 'http://example.com']]);
 xpath
--------
 {test}
(1 row)
```

xpath_exists(xpath, xml [, nsarray])

描述：函数xpath_exists是xpath函数的一种特殊形式。这个函数不是返回满足XPath的XML值，它返回一个布尔值表示查询是否被满足。这个函数等价于标准的XMLEXISTS谓词，不过它还提供了对一个名字空间映射参数的支持。

返回值：bool

示例：

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
ARRAY[ARRAY['my', 'http://example.com']]);
 xpath_exists
--------------
 t
(1 row)
```


xmltable

描述：xmltable函数根据输入的XML数据、XPath路径表达式、列的定义信息生成一个表。xmltable在语法上类似于一个函数，但它只能以表的形式出现在查询的FROM子句里。

返回值：setof record

语法：

```
XMLTABLE ( [ XMLNAMESPACES ( namespace_uri AS namespace_name [,  ...] ), ]
           row_expression PASSING [ BY  { REF | VALUE } ]
document_expression [ BY  { REF | VALUE } ]
COLUMNS name  { type  [ PATH column_expression  ] [ DEFAULT default_expression ] [ NOT NULL |
NULL ] | FOR ORDINALITY }
[, ...]
)
```

参数说明：

- 可选的XMLNAMESPACES子句是一个逗号分隔的命名空间定义列表，其中namespace_uri都是text类型表达式，namespace_name都是简单标识符。XMLNAMESPACES指定了在文档中使用的XML命名空间及它们的别名。当前不支持默认的命名空间声明。
- 必选的row_expression参数是一个XPath 1.0表达式，该表达式根据提供的XML文档document_expression进行计算获取XML节点序列，该序列是xmltable转换成输出行的顺序。如果document_expression值为NULL，或row_expression产生空的节点集，结果将不返回任何行。
- document_expression参数用于输入XML文档，输入的文档必须是符合XML格式的文档，不接受XML片段数据或格式错误的XML文档。BY REF和BY VALUE子句不起作用，仅用于实现SQL标准兼容性。
- COLUMNS子句指定输出表中字段列表定义，列名和列的数据类型是必选的，路径、默认值和是否为空子句是可选的。
  - 列的column_expression是一个XPath 1.0表达式，用于从row_expression计算出当前行中提取出列的值。如果没有给出column_expression，那么字段名将用作隐式路径。
  - 列可以被标记为NOT NULL，如果NOT NULL列的column_expression不返回任何数据，并且没有DEFAULT子句或者default_expression的计算结果为NULL，就会报错。
  - 标记为FOR ORDINALITY的字段将从1开始填充行号，其顺序为从row_expression结果集中检索到的节点顺序，最多只有一列可以标记为FOR ORDINALITY。

  须知

  XPath 1.0没有为节点指定顺序，因此返回结果的顺序取决于数据获取顺序。

示例：

```
SELECT * FROM XMLTABLE('/ROWS/ROW'
PASSING '<ROWS><ROW id="1"><CITY_ID>1002a</CITY_ID><CITY_NAME>snowcity</CITY_NAME></
ROW><ROW id="2"><CITY_ID>1003b</CITY_ID><CITY_NAME>icecity</CITY_NAME></ROW><ROW
id="3"><CITY_ID>1004c</CITY_ID><CITY_NAME>windcity</CITY_NAME></ROW></ROWS>'
COLUMNS id INT PATH '@id',
ordinality FOR ORDINALITY,
```



```
CITY_id TEXT PATH 'CITY_ID',CITY_name TEXT PATH 'CITY_NAME' NOT NULL);
 id | ordinality | city_id | city_name
----+------------+---------+-----------
  1 |          1 | 1002a   | snowcity
  2 |          2 | 1003b   | icecity
  3 |          3 | 1004c   | windcity
(3 rows)
```

6.32.4 将表映射到 XML

本节的函数将会把关系表的内容映射成XML值。可以认为是XML导出功能。

将表映射到XML函数的参数说明如下：

- tbl：表名。
- nulls：在输出中是否包含空值，若为true，列中的空值表示为：<columnname xsi:nil="true"/>，若为false，包含空值的列会从输出中省略。
- tableforest：若为true，则输出xml片段，false，输出xml文档。
- targetns：指定想要结果的XML命名空间。若不指定，应传递一个空字符串。
- query：SQL查询语句。
- cursor：游标名。
- count：从游标中获取的数据量。
- schema：模式名

table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)

描述：把表的内容映射成XML值。

返回值类型：xml

如果tableforest为假，则结果的XML文档类似如下：

```
<tablename>
 <row>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
 </row>

 <row>
  ...
 </row>

 ...
</tablename>
```

如果tableforest为真，则结果的XML内容片断类似如下：

```
<tablename>
 <columnname1>data</columnname1>
 <columnname2>data</columnname2>
</tablename>

<tablename>
 ...
</tablename>

...
```


table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

描述：把关系表的模式映射成XML模式文档。

返回值类型：xml

模式内容映射的结果可能类似如下：

```
<schemaname>
table1-mapping
table2-mapping
...
</schemaname>
```

其中表映射的格式取决于tableforest参数。

table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)

描述：把关系表映射成XML值和模式文档。

返回值类型：xml

query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)

描述：把SQL查询的内容映射成XML值。

返回值类型：xml

query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

描述：把SQL查询映射成XML模式文档。

返回值类型：xml

query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

描述：把SQL查询映射成XML值和模式文档。

返回值类型：xml

cursor_to_xml(cursor refcursor, count int, nulls boolean, tableforest boolean, targetns text)

描述：把游标查询映射成XML值。

返回值类型：xml

cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)

描述：把游标查询映射成XML模式文档。

返回值类型：xml

schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)

描述：把模式中的表映射成XML值。

返回值类型：xml

schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

描述：把模式中的表映射成XML模式文档

返回值类型：xml

schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

描述：把模式中的表映射成XML值和模式文档。

返回值类型：xml

database_to_xml(nulls boolean, tableforest boolean, targetns text)

描述：把数据库的表映射成XML值。

返回值类型：xml

数据库内容映射的结果可能类似如下：

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)

描述：把数据库的表映射成XML模式文档

返回值类型：xml

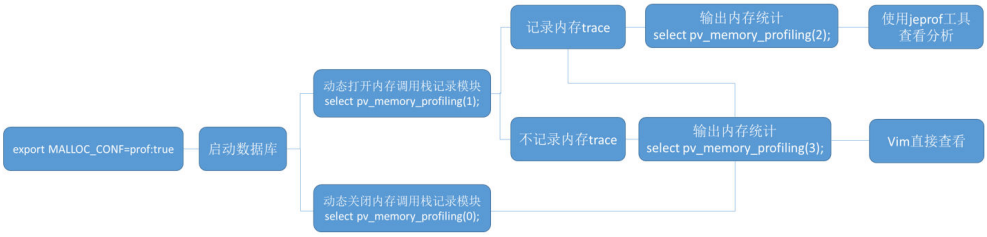
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)

描述：把数据库的表映射成XML值和模式文档。

返回值类型：xml

6.33 调用栈记录函数

GaussDB(DWS)通过pv_memory_profiling(type int)和环境变量MALLOC_CONF，控制malloc等内存分配调用栈记录模块开启关闭、以及内存调用栈输出等，使用流程如下图所示：



MALLOC_CONF

环境变量MALLOC_CONF用于控制监控模块是否开启，位于${BIGDATA_HOME}/mppdb/.mppdbgs_profile文件内，默认开启。需注意：

- 修改此环境变量变化，需要重启数据库。
- 如果在集群中启用了om_monitor，完成环境变量设置后，先重启om_monitor进程后，然后重启数据库，使得开关生效。
- 该环境变量可以设置在集群所有服务器中，也可以仅设置在需要开启模块的个别服务器中，对GaussDB进程而言，各进程是根据各自MALLOC_CONF环境变量，控制模块是否打开。

MALLOC_CONF开启和关闭命令：

- 开启监控模块：
  ```
  export MALLOC_CONF=prof:true
  ```
- 关闭监控模块：
  ```
  export MALLOC_CONF=prof:false
  ```

pv_memory_profiling (type int)

参数说明：控制内核malloc等内存分配函数的backtrace记录及输出。

取值范围：正整数，0~3

表 6-43 pv_memory_profiling 取值含义

pv_memory_profiling 取值	含义
0	关闭内存trace，不记录malloc等调用栈信息。
1	开启内存trace，记录malloc等调用栈信息。
2	输出malloc等调用栈的trace日志。 ● 输出路径为：/proc/pid/cwd指向的目录，pid为gaussdb进程ID。 ● 输出日志命名规则为：jeprof.<pid>.*.heap，其中pid表示GaussDB进程ID，*表示trace日志输出的序号，序号唯一。如：jeprof.195473.0.u0.heap
3	输出内存统计信息。 ● 输出路径为：/proc/pid/cwd指向的目录，pid为gaussdb进程ID。 ● 命名规则为节点名称+进程id+时间+“heap_stats”+.out。该文件可用vim直接打开。

返回值类型：boolean

备注：

- 如果成功，函数返回true，否则返回false，返回相关提示信息。
- 如果失败提示信息为“Memory profiling failed, check if $MALLOC_CONF contain 'prof:true'.”，说明未设置MALLOC_CONF=prof:true的情况下，使用了本模块，需要设置环境变量即可解决。
- 如果失败提示信息为“Type %d is not supported. The valid range is 0-3.”，说明用户输入参数错误，正确数值为0,1,2,3。
- 如果失败提示信息为“Memory profiling failed, inputed type is %d, failed number is %d.”请联系技术支持工程师提供技术支持。

输出内存调用栈信息

操作步骤：

步骤1 输出内存调用栈信息，在GaussDB进程所在目录输出trace文件：

```
SELECT * FROM pv_memory_profiling(2);
```

步骤2 使用jemalloc中提供的jeprof工具，解析日志信息：

方式1：以text格式输出。

```
jeprof --text --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 >prof.txt
```

方式2：以pdf格式输出。

```
jeprof --pdf --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 > prof.pdf
```


说明

- 解析内存调用栈信息，需要依靠GaussDB源码进行分析，需要将trace文件返回给研发工程师，进行分析。
- 分析trace文件，需要使用jeprof工具，该工具由Jemalloc生成。在常规使用中，需要依赖perl环境，如果需要生成pdf调用图，需要安装与操作系统匹配的GraphViz工具。

----结束

示例

```
--系统管理员用户登录，设置环境变量，启动数据库。
export MALLOC_CONF=prof:true

--在数据库运行期间，关闭内存trace记录功能。
SELECT pv_memory_profiling(0);
pv_memory_profiling
----------------------------
t
(1 row)

--在数据库运行期间，打开内存trace记录功能。
SELECT pv_memory_profiling(1);
pv_memory_profiling
----------------------------
t
(1 row)

--输出内存trace记录功能。
SELECT pv_memory_profiling(2);
pv_memory_profiling
----------------------------
t
(1 row)

--在GaussDB进程所在目录输出trace文件，以text格式或PDF格式输出。
jeprof --text --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 >prof.txt
jeprof --pdf --show_bytes $GAUSSHOME/bin/gaussdb trace文件1 > prof.pdf

--输出内存统计信息。
执行以下命令，在GaussDB进程所在目录输出内存统计文件，可直接读取。
SELECT pv_memory_profiling(3);
pv_memory_profiling
----------------------------
t
(1 row)
```


7 表达式

7.1 简单表达式

逻辑表达式

逻辑表达式的操作符和运算规则，请参见**逻辑操作符**。

比较表达式

常用的比较操作符，请参见**比较操作符**。

除比较操作符外，还可以使用以下句式结构：

- BETWEEN操作符

  操作符BETWEEN...AND会选取介于两个值之间的数据范围。这些值可以是数值、文本或日期。

  a BETWEEN x AND y表达式等效于a >= x AND a <= y表达式

```
SELECT 2 BETWEEN 1 AND 3 AS RESULT;
 result
----------
 t
(1 row)

SELECT 2 >= 1 AND 2 <= 3 AS RESULT;
 result
----------
 t
(1 row)
```

  a NOT BETWEEN x AND y等效于a < x OR a > y

```
SELECT 2 NOT BETWEEN 1 AND 3 AS RESULT;
 result
----------
 f
(1 row)

SELECT 2 < 1 OR 2 > 3 AS RESULT;
 result
----------
 f
(1 row)
```

- 检查一个值是不是null，可使用：

```
expression IS NULL
expression IS NOT NULL
SELECT 2+2 IS NULL AS RESULT;
 result
----------
 f
(1 row)

SELECT 2+2 IS NOT NULL AS RESULT;
 result
----------
 t
(1 row)
```

或者与之等价的句式结构，但不是标准的：

```
expression   ISNULL
expression   NOTNULL
```

须知

不要写expression=NULL或expression<>(!=)NULL，因为NULL代表一个未知的值，不能通过该表达式判断两个未知值是否相等。

```
SELECT 2+2 ISNULL AS RESULT;
 result
----------
 f
(1 row)

SELECT 2+2 NOTNULL AS RESULT;
 result
----------
 t
(1 row)
SELECT 2+2 IS DISTINCT FROM NULL AS RESULT;
 result
----------
 t
(1 row)

SELECT 2+2 IS NOT DISTINCT FROM NULL AS RESULT;
 result
----------
 f
(1 row)
```

7.2 条件表达式

在执行SQL语句时，可通过条件表达式筛选出符合条件的数据。

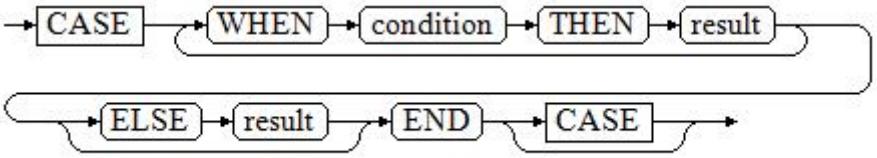
条件表达式主要有以下几种：

- CASE

  CASE表达式是条件表达式，类似于其他编程语言中的CASE语句。

  CASE表达式的语法图请参考**图7-1**。

图 7-1 case::=



CASE子句可以用于合法的表达式中。condition是一个返回BOOLEAN数据类型的表达式：

- 如果结果为真，CASE表达式的结果就是符合该条件所对应的result。
- 如果结果为假，则以相同方式处理随后的WHEN或ELSE子句。
- 如果各WHEN condition都不为真，表达式的结果就是在ELSE子句执行的result。如果省略了ELSE子句且没有匹配的条件，结果为NULL。

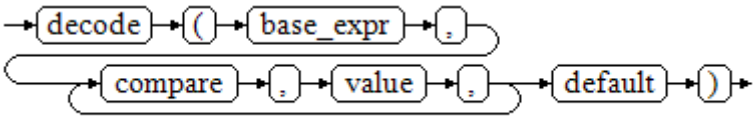
示例：

```
CREATE TABLE tpcds.case_when_t1(CW_COL1 INT)  DISTRIBUTE BY HASH (CW_COL1);

INSERT INTO tpcds.case_when_t1 VALUES (1), (2), (3);

SELECT * FROM tpcds.case_when_t1;
 cw_col1
---------
       3
       1
       2
(3 rows)

SELECT CW_COL1, CASE WHEN CW_COL1=1 THEN 'one' WHEN CW_COL1=2 THEN 'two' ELSE 'other'
END FROM tpcds.case_when_t1;
 cw_col1 | case
---------+-------
       3 | other
       1 | one
       2 | two
(3 rows)

DROP TABLE tpcds.case_when_t1;
```

- DECODE

  DECODE的语法图请参见**图7-2**。

图 7-2 decode::=



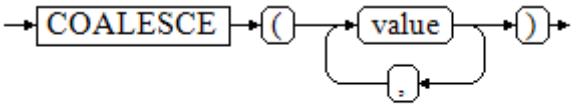
将表达式base_expr与后面的每个compare(n)进行比较，如果匹配返回相应的value(n)。如果没有发生匹配，则返回default。

示例请参见**条件表达式函数**。

```
SELECT DECODE('A','A',1,'B',2,0);
 case
------
    1
(1 row)
```

- COALESCE
  COALESCE的语法图请参见**图7-3**。

  图 7-3 coalesce::=

  COALESCE返回它的第一个非NULL的参数值。如果参数都为NULL，则返回NULL。它常用于在显示数据时用缺省值替换NULL。和CASE表达式一样，COALESCE只计算用来判断结果的参数，即在第一个非空参数右边的参数不会被计算。

  示例：

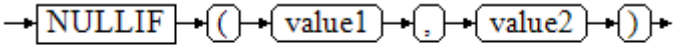
```
CREATE TABLE tpcds.c_tabl(description varchar(10), short_description varchar(10), last_value
varchar(10))
DISTRIBUTE BY HASH (last_value);

INSERT INTO tpcds.c_tabl VALUES('abc', 'efg', '123');
INSERT INTO tpcds.c_tabl VALUES(NULL, 'efg', '123');

INSERT INTO tpcds.c_tabl VALUES(NULL, NULL, '123');

SELECT description, short_description, last_value, COALESCE(description, short_description, last_value)
FROM tpcds.c_tabl ORDER BY 1, 2, 3, 4;
 description | short_description | last_value | coalesce
-------------+-------------------+------------+----------
 abc         | efg               | 123        | abc
             | efg               | 123        | efg
             |                   | 123        | 123
(3 rows)

DROP TABLE tpcds.c_tabl;
```

  如果description不为NULL，则返回description的值，否则计算下一个参数short_description；如果short_description不为NULL，则返回short_description的值，否则计算下一个参数last_value；如果last_value不为NULL，则返回last_value的值，否则返回（none）。

```
SELECT COALESCE(NULL,'Hello World');
   coalesce
---------------
 Hello World
(1 row)
```

- NULLIF
  NULLIF的语法图请参见**图7-4**。

  图 7-4 nullif::=

  只有当value1和value2相等时，NULLIF才返回NULL。否则它返回value1。

  示例：

```
CREATE TABLE tpcds.null_if_t1 (
    NI_VALUE1 VARCHAR(10),
```



```
    NI_VALUE2 VARCHAR(10)
)  DISTRIBUTE BY HASH (NI_VALUE1);

INSERT INTO tpcds.null_if_t1 VALUES('abc', 'abc');
INSERT INTO tpcds.null_if_t1 VALUES('abc', 'efg');

SELECT NI_VALUE1, NI_VALUE2, NULLIF(NI_VALUE1, NI_VALUE2) FROM tpcds.null_if_t1 ORDER BY 1,
2, 3;

 ni_value1 | ni_value2 | nullif
-----------+-----------+--------
 abc       | abc       |
 abc       | efg       | abc
(2 rows)
DROP TABLE tpcds.null_if_t1;
```

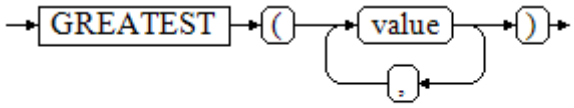
如果value1等于value2则返回NULL，否则返回value1。

```
SELECT NULLIF('Hello','Hello World');
 nullif
--------
 Hello
(1 row)
```

- GREATEST（最大值），LEAST（最小值）

  GREATEST的语法图请参见图7-5。

  图 7-5 greatest::=

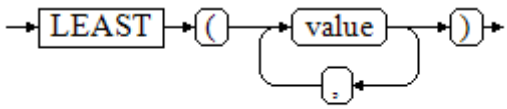
  

  从一个任意数字表达式的列表里选取最大的数值。

```
SELECT greatest(9000,155555,2.01);
 greatest
----------
   155555
(1 row)
```

  LEAST的语法图请参见图7-6。

  图 7-6 least::=

  从一个任意数字表达式的列表里选取最小的数值。

  以上的数字表达式必须都可以转换成一个普通的数据类型，该数据类型将是结果类型。

  列表中的NULL值将被忽略。只有所有表达式的结果都是NULL的时候，结果才是NULL。

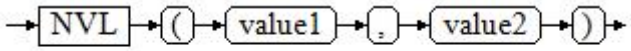
```
SELECT least(9000,2);
 least
-------
     2
(1 row)
```

示例请参见**条件表达式函数**。

- NVL

  NVL的语法图请参见**图7-7**。

  图 7-7 nvl::=

  如果value1为NULL则返回value2，如果value1非NULL，则返回value1。

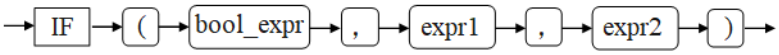
  示例：

  ```
  SELECT nvl(null,1);
  nvl
  -----
   1
  (1 row)
  SELECT nvl ('Hello World' ,1);
       nvl
  ---------------
   Hello World
  (1 row)
  ```

- IF

  IF的语法图请参见**图7-8**。

  图 7-8 if::=

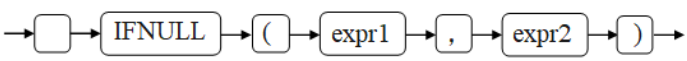
  当bool_expr为true时，返回expr1，否则返回expr2。

  示例请参见**条件表达式函数**。

- IFNULL

  IFNULL的语法图请参见**图7-9**。

  图 7-9 ifnull::=

  只有当value1和value2相等时，IFNULL才返回NULL。否则它返回value1。

  示例请参见**条件表达式函数**。

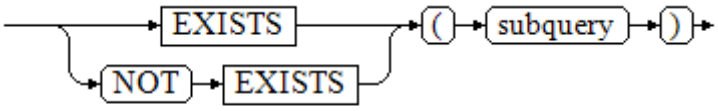
7.3 子查询表达式

子查询表达式主要有以下几种：

- EXISTS/NOT EXISTS

  EXISTS/NOT EXISTS的语法图请参见图7-10。

  图 7-10 EXISTS/NOT EXISTS::=

  EXISTS的参数是一个任意的SELECT语句或者子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，则EXISTS结果就为true；如果子查询没有返回任何行， EXISTS的结果为false。

  EXISTS/NOT EXISTS子查询通常只是运行到能判断它是否可以生成至少一行为止，而不是等到全部结束。

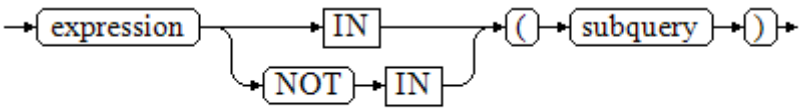
  示例：

```
SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE EXISTS (SELECT d_dom FROM
tpcds.date_dim WHERE d_dom = store_returns.sr_reason_sk and sr_customer_sk <10);
sr_reason_sk | sr_customer_sk
--------------+----------------
          13 |              2
          22 |              5
          17 |              7
          25 |              7
           3 |              7
          31 |              5
           7 |              7
          14 |              6
          20 |              4
           5 |              6
          10 |              3
           1 |              5
          15 |              2
           4 |              1
          26 |              3
(15 rows)
```

- IN/NOT IN

  IN/NOT IN的语法请参见图7-11。

  图 7-11 IN/NOT IN::=

  右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式对子查询结果的每一行进行一次计算和比较。如果找到任何相等的子查询行，则IN结果

为true。如果没有找到任何相等行，则结果为false（包括子查询没有返回任何行的情况）。

表达式或子查询行里的NULL遵照SQL处理布尔值和NULL组合时的规则。如果两个行对应的字段都相等且非空，则这两行相等；如果任意对应字段不等且非空，则这两行不等；否则结果是未知（NULL）。如果每一行的结果都是不等或NULL，并且至少有一个NULL，则IN的结果是NULL。

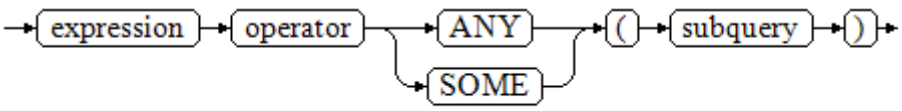
示例：

```
 SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk IN (SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
--------------+----------------
          10 |              3
          26 |              3
          22 |              5
          31 |              5
           1 |              5
          32 |              5
          32 |              5
           4 |              1
          15 |              2
          13 |              2
          33 |              4
          20 |              4
          33 |              8
           5 |              6
          14 |              6
          17 |              7
           3 |              7
          25 |              7
           7 |              7
(19 rows)
```

- ANY/SOME

  ANY/SOME的语法图请参见**图7-12**。

  图 7-12 any/some::=

  右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式使用operator对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如果至少获得一个真值，则ANY结果为true。如果全部获得假值，则结果为false（包括子查询没有返回任何行的情况）。SOME是ANY的同义词。IN与ANY可以等效替换。

  示例：

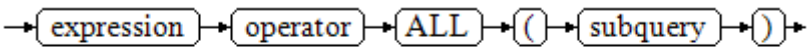
```
SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk < ANY (SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
--------------+----------------
          26 |              3
          17 |              7
          32 |              5
          32 |              5
          13 |              2
          31 |              5
          25 |              7
```

```
          5 |           6
          7 |           7
         10 |           3
          1 |           5
         14 |           6
          4 |           1
          3 |           7
         22 |           5
         33 |           4
         20 |           4
         33 |           8
         15 |           2
(19 rows)
```

- ALL

  ALL的语法请参见图7-13。

  图 7-13 all::=

  右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式使用operator对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如果全部获得真值，ALL结果为true（包括子查询没有返回任何行的情况）。如果至少获得一个假值，则结果为false。

  示例：

```
SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk < all(SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
 sr_reason_sk | sr_customer_sk
--------------+----------------
(0 rows)
```

7.4 数组表达式

IN

expression **IN** *(value [, ...])*

右侧括号中的是一个表达式列表。左侧表达式的结果与表达式列表的内容进行比较。如果列表中的内容符合左侧表达式的结果，则IN的结果为true。如果没有相符的结果，则IN的结果为false。

示例如下：

```
SELECT 8000+500 IN (10000, 9000) AS RESULT;
 result
----------
 f
(1 row)
```

说明

如果表达式结果为null，或者表达式列表不符合表达式的条件且右侧表达式列表返回结果至少一处为空，则IN的返回结果为null，而不是false。这样的处理方式和SQL返回空值的布尔组合规则是一致的。

NOT IN

expression **NOT IN** *(value [, ...])*

右侧括号中的是一个表达式列表。左侧表达式的结果与表达式列表的内容进行比较。如果在列表中的内容没有符合左侧表达式结果的内容，则NOT IN的结果为true。如果有符合的内容，则NOT IN的结果为false。

示例如下：

```
SELECT 8000+500 NOT IN (10000, 9000) AS RESULT;
 result
----------
 t
(1 row)
```

说明

- 如果查询语句返回结果为空，或者表达式列表不符合表达式的条件且右侧表达式列表返回结果至少一处为空，则NOT IN的返回结果为null，而不是false。这样的处理方式和SQL返回空值的布尔组合规则是一致的。
- 在所有情况下X NOT IN Y等价于NOT(X IN Y)。

ANY/SOME (array)

expression operator **ANY** *(array expression)*

expression operator **SOME** *(array expression)*

```
SELECT 8000+500 < SOME (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
SELECT 8000+500 < ANY (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
```

右侧括号中的是一个数组表达式，它必须产生一个数组值。左侧表达式的结果使用操作符对数组表达式的每一行结果都进行计算和比较，比较结果必须是布尔值。

- 如果对比结果至少获取一个真值，则ANY的结果为true。
- 如果对比结果没有真值，则ANY的结果为false。

说明

- 如果结果没有真值，并且数组表达式生成至少一个值为null，则ANY的值为NULL，而不是false。这样的处理方式和SQL返回空值的布尔组合规则是一致的。
- SOME是ANY的同义词。

ALL (array)

expression operator **ALL** *(array expression)*

右侧括号中的是一个数组表达式，它必须产生一个数组值。左侧表达式的结果使用操作符对数组表达式的每一行结果都进行计算和比较，比较结果必须是布尔值。

- 如果所有的比较结果都为真值（包括数组不含任何元素的情况），则ALL的结果为true。

- 如果结果有任一结果是false，则ALL的结果为false。

如果数组表达式产生一个NULL数组，则ALL的结果为NULL。如果左边表达式的值为NULL ，则ALL的结果通常也为NULL(某些不严格的比较操作符可能得到不同的结果)。另外，如果右边的数组表达式中包含null元素并且比较结果没有假值，则ALL的结果将是NULL(某些不严格的比较操作符可能得到不同的结果)， 而不是真。这样的处理方式和SQL返回空值的布尔组合规则是一致的。

```
SELECT 8000+500 < ALL (array[10000,9000]) AS RESULT;
 result
----------
 t
(1 row)
```

7.5 行表达式

语法：

```
row_constructor operator row_constructor
```

两边都是一个行构造器，两行值必须具有相同数目的字段，每一行都进行比较，行比较允许使用=，<>，<，<=，>=等操作符，或其中一个相似的语义符。

=<>和别的操作符使用略有不同。如果两行值的所有字段都是非空并且相等，则认为两行是相等的；如果两行值的任意字段为非空并且不相等，则认为两行是不相等的；否则比较结果是未知的（null）。

对于<，<=，>，> =的情况下，行中元素从左到右依次比较，直到遇到一对不相等的元素或者一对为空的元素。如果这对元素中存在至少一个null值，则比较结果是未知的（null），否则这对元素的比较结果为最终的结果。

示例：

```
SELECT ROW(1,2,NULL) < ROW(1,3,0) AS RESULT;
 result
----------
 t
(1 row)
```

8 类型转换

8.1 概述

背景信息

在SQL语言中，每个数据都与一个决定其行为和用法的数据类型相关。GaussDB(DWS)提供一个可扩展的数据类型系统，该系统比其它SQL实现更具通用性和灵活性。因而，GaussDB(DWS)中大多数类型转换是由通用规则来管理的，这种做法允许使用混合类型的表达式。

GaussDB(DWS)扫描/分析器只将词法元素分解成五个基本种类：整数、浮点数、字符串、标识符和关键字。大多数非数字类型首先表现为字符串。SQL语言的定义允许将常量字符串声明为具体的类型。例如：

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";
 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

示例中有两个文本常量，类型分别为text和point。如果没有为字符串文本声明类型，则该文本首先被定义成一个unknown类型。

在GaussDB(DWS)分析器里，有四种基本的SQL结构需要独立的类型转换规则：

- 函数调用

  多数SQL类型系统是建筑在一套丰富的函数上的。函数调用可以有一个或多个参数。因为SQL允许函数重载，所以不能通过函数名直接找到要调用的函数，分析器必须根据函数提供的参数类型选择正确的函数。

- 操作符

  SQL允许在表达式上使用前缀或后缀（单目）操作符，也允许表达式内部使用双目操作符（两个参数）。像函数一样，操作符也可以被重载，因此操作符的选择也和函数一样取决于参数类型。

- 值存储

  INSERT和UPDATE语句将表达式结果存入表中。语句中的表达式类型必须和目标字段的类型一致或者可以转换为一致。

- UNION，CASE和相关构造

因为联合SELECT语句中的所有查询结果必须在一列里显示出来，所以每个SELECT子句中的元素类型必须相互匹配并转换成一个统一类型。类似地，一个CASE构造的结果表达式必须转换成统一的类型，这样整个case表达式会有一个统一的输出类型。同样的要求也存在于ARRAY构造以及GREATEST和LEAST函数中。

系统表PG_CAST存储了有关数据类型之间的转换关系以及如何执行这些转换的信息。

语义分析阶段会决定表达式的返回值类型并选择适当的转换行为。数据类型的基本类型分类，包括：boolean、numeric、string、bitstring、datetime、timespan、geometric和network。每种类型都有一种或多种首选类型用于解决类型选择的问题。根据首选类型和可用的隐含转换，就可能保证有歧义的表达式（那些有多个候选解析方案的）得到有效的方式解决。

所有类型转换规则需遵循以下基本原则：

- 隐含转换不能有奇怪的或不可预见的输出。
- 如果一个查询不需要隐含的类型转换，分析器和执行器不应该进行更多的额外操作。即任何一个类型匹配、格式清晰的查询不应该在分析器里耗费更多的时间，也不应该向查询中引入任何不必要的隐含类型转换调用。
- 如果一个查询在调用某个函数时需要进行隐式转换，当用户定义了一个有正确参数的函数后，解释器应该选择使用新函数。

TD 兼容模式下，空串转换为数值类型的处理

- TD数据库不同于Oracle，Oracle将空串当做NULL进行处理，TD在将空串转换为数值类型的时候，默认将空串转换为0进行处理，因此查询空串会查询到数值为0的数据。同样地，在TD兼容模式下，字符串转换数值的过程中，也会将空串默认转换为相应数值类型的0值进行处理。除此之外，' - '、' + '、' '这些字符串也都会在TD兼容模式下默认转换为0进行处理，但是小数点字符串' . '会报错。例如：

```
CREATE TABLE t1(no int,col varchar);
INSERT INTO t1 values(1,'');
INSERT INTO t1 values(2,null);
SELECT * FROM t1 WHERE col is null;
 no | col
----+-----
  2 |
(1 row)

SELECT * FROM t1 WHERE col='';
 no | col
----+-----
 1  |
(1 row)
```

- MySQL兼容模式下对空串转换为数值类型的处理和TD兼容模式相同。

8.2 操作符

操作符类型解析

1. 从系统表PG_OPERATOR中选出要考虑的操作符。如果可以找到一个参数类型以及参数个数都一致的操作符，那么这个操作符就是最终使用的操作符。如果找到了多个备选的操作符，那么将从中选择一个最合适的。
2. 寻找最优匹配。
   a. 丢弃输入类型不匹配以及无法隐式转换成匹配的候选操作符。unknown文本在这种情况下可以转换成任何类型。如果只剩下一个候选操作符，则使用，否则继续下一步。

b. 查看所有候选操作符，并保留输入类型最匹配的操作符。此时，域被看作和其基本类型相同。如果没有完全匹配的操作符，则保留所有候选。如果只剩下一个候选操作符，则使用，否则继续下一步。

c. 查看所有候选操作符，保留需要类型转换时接受（属于输入数据类型的类型范畴的）首选类型位置最多的操作符。如果没有接受首选类型的操作符，则保留所有候选。如果只剩下一个候选操作符，则使用，否则继续下一步。

d. 如果有任何输入参数是unknown类型，请检查其余候选操作符对应参数位置的类型范畴。在每一个能够接受string类型范畴的位置使用string类型（这种偏向字符串的做法合理，因为unknown文本跟字符串相似）。另外，如果所有剩下的候选操作符都接受相同的类型范畴，则选择该类型范畴，否则会报错（因为在没有更多线索的条件下无法作出正确的选择）。现在丢弃不接受选定类型范畴的候选操作符。此外，如果有任意候选操作符接受该范畴中的首选类型，则丢弃该参数接受非首选类型的候选操作符。如果没有一个操作符能被保留，则保留所有候选。如果只剩下一个候选操作符，则使用，否则继续下一步。

e. 如果同时有unknown和已知参数，并且所有已知参数都是相同的类型，那么假设unknown参数也属于该类型，并检查哪些候选操作符在unknown参数位置接受该类型。如果只有一个操作符符合，则使用。否则，报错。

示例

示例1：阶乘操作符类型解析。在系统表中里只有一个阶乘操作符（后缀!），它以bigint作为参数。扫描器给下面查询表达式的参数赋予bigint的初始类型：

```
SELECT 40 ! AS "40 factorial";

                  40 factorial
-----------------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

分析器对参数做类型转换，查询等效于：

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

示例2：字符串连接操作符类型分析。一种字符串风格的语法既可以用于字符串也可以用于复杂的扩展类型。未声明类型的字符串将被所有可能的候选操作符匹配。有一个未声明的参数的例子：

```
SELECT text 'abc' || 'def' AS "text and unknown";
 text and unknown
------------------
 abcdef
(1 row)
```

本例中分析器寻找两个参数都是text的操作符。确实有这样的操作符，两个参数都是text类型。

下面是连接两个未声明类型的值：

```
SELECT 'abc' || 'def' AS "unspecified";
 unspecified
-------------
 abcdef
(1 row)
```

说明

因为查询中没有声明任何类型，所以本例中对类型没有任何初始提示。因此，分析器查找所有候选操作符，发现既存在接受字符串类型范畴的操作符也存在接受位串类型范畴的操作符。因为字符串类型范畴是首选，所以选择字符串类型范畴的首选类型text作为解析未知类型文本的声明类型。

示例3：绝对值和取反操作符类型分析。GaussDB(DWS)操作符表里面有几条记录对应于前缀操作符@，它们都用于为各种数值类型实现绝对值操作。其中之一用于float8类型，它是数值类型范畴中的首选类型。因此，在面对unknown输入的时候，GaussDB(DWS)会使用该类型：

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

此处，系统在应用选定的操作符之前隐式的转换unknown类型的文字为float8类型。

示例4：数组包含操作符类型分析。以操作符两侧带有一个已知类型和一个未知类型的情况为例：

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";
 is subset
-----------
 t
(1 row)
```

说明

GaussDB(DWS)的pg_operator表中有多条记录对应于中缀操作符<@，但是只有“数组包含(anyarray <@ anyarray)”和“范围包含(anyelement <@ anyrange)”可在左侧接受一个整数数组。因为多态伪类型(参阅**伪类型**)不被认为是首选操作符类型，因此解析器无法在此基础上解决歧义。不过，**寻找最优匹配**的最后一条解析规则又指出，假定未知类型的文字和另一个已知的输入类型相同，也就是只有两个运算符之一可以匹配，所以选择数组包含。（如果选择了范围包含，就会报错，因为字符串格式不正确不能作为范围。）

8.3 函数

函数类型解析

1. 从系统表PG_PROC中选择所有可能被选到的函数。如果使用了一个不带模式修饰的函数名字，那么认为该函数是那些在当前搜索路径中的函数。如果给出一个带修饰的函数名，那么只考虑指定模式中的函数。

   如果搜索路径中找到了多个不同参数类型的函数。将从中选择一个合适的函数。
2. 查找和输入参数类型完全匹配的函数。如果找到一个，则用之。如果输入的实参类型都是unknown类型，则不会找到匹配的函数。
3. 如果未找到完全匹配，请查看该函数是否为一个特殊的类型转换函数。
4. 寻找最优匹配。
   a. 抛弃那些输入类型不匹配并且也不能隐式转换成匹配的候选函数。unknown文本在这种情况下可以转换成任何东西。如果只剩下一个候选项，则用之，否则继续下一步。
   b. 遍历所有候选函数，保留那些输入类型匹配最准确的。此时，域被看作和它们的基本类型相同。如果没有一个函数能准确匹配，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

c. 遍历所有候选函数，保留那些需要类型转换时接受首选类型位置最多的函数。如果没有接受首选类型的函数，则保留所有候选。如果只剩下一个候选项，则用之，否则继续下一步。

d. 如果有任何输入参数是unknown类型，检查剩余的候选函数对应参数位置的类型范畴。在每一个能够接受字符串类型范畴的位置使用string类型（这种对字符串的偏爱是合适的，因为unknown文本确实像字符串）。另外，如果所有剩下的候选函数都接受相同的类型范畴，则选择该类型范畴，否则抛出一个错误（因为在没有更多线索的条件下无法作出正确的选择）。现在抛弃不接受选定的类型范畴的候选函数，然后，如果任意候选函数在那个范畴接受一个首选类型，则抛弃那些在该参数位置接受非首选类型的候选函数。如果没有一个候选符合这些测试则保留所有候选。如果只有一个候选函数符合，则使用它；否则，继续下一步。

e. 如果同时有unknown和已知类型的参数，并且所有已知类型的参数有相同的类型，假设unknown参数也是这种类型，检查哪个候选函数可以在unknown参数位置接受这种类型。如果正好一个候选符合，那么使用它。否则，产生一个错误。

示例

示例1：圆整函数参数类型解析。只有一个round函数有两个参数（第一个是numeric，第二个是integer）。所以下面的查询自动把第一个类型为integer的参数转换成numeric类型。

```
SELECT round(4, 4);
 round
--------
 4.0000
(1 row)
```

实际上它被分析器转换成：

```
SELECT round(CAST (4 AS numeric), 4);
```

因为带小数点的数值常量初始时被赋予numeric类型，因此下面的查询将不需要类型转换，并且可能会略微高效一些：

```
SELECT round(4.0, 4);
```

示例2：子字符串函数类型解析。有好几个substr函数，其中一个接受text和integer类型。如果用一个未声明类型的字符串常量调用它，系统将选择接受string类型范畴的首选类型（也就是text类型）的候选函数。

```
SELECT substr('1234', 3);
 substr
--------
 34
(1 row)
```

如果该字符串声明为varchar类型，就像从表中取出来的数据一样，分析器将试着将其转换成text类型：

```
SELECT substr(varchar '1234', 3);
 substr
--------
    34
(1 row)
```

被分析器转换后实际上变成：

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```


说明

分析器从PG_CAST表中获取到text和varchar是二进制兼容的，即可传递给接受类型的函数而不需要做任何物理转换。因此，在这种情况下，实际上没有做任何类型转换。

而且，如果以integer为参数调用函数，分析器将试图将其转换成text类型：

```
SELECT substr(1234, 3);
substr
--------
 34
(1 row)
```

被分析器转换后实际上变成：

```
SELECT substr(CAST (1234 AS text), 3);
 substr
--------
 34
(1 row)
```

8.4 值存储

值存储数据类型解析

1. 查找与目标字段准确的匹配。
2. 试着将表达式直接转换成目标类型。如果已知这两种类型之间存在一个已登记的转换函数，那么直接调用该转换函数即可。如果表达式是一个未知类型文本，该文本字符串的内容将交给目标类型的输入转换过程。
3. 检查目标类型是否有长度转换。长度转换是一个从某类型到自身的转换。如果在pg_cast表里面找到一个，那么在存储到目标字段之前先在表达式上应用。这样的转换函数总是接受一个额外的类型为integer的参数，它接收目标字段的atttypmod值（实际上是其声明长度，atttypmod的解释随不同的数据类型而不同），并且它可能接受一个boolean类型的第三个参数，表示转换是显式的还是隐式的。转换函数负责施加那些长度相关的语义，比如长度检查或者截断。

示例

character存储类型转换。对一个目标列定义为character(20)的语句，下面的语句显示存储值的长度正确：

```
CREATE TABLE x1
(
    customer_sk             integer,
    customer_id             char(20),
    first_name              char(6),
    last_name               char(8)
)
with (orientation = column,compression=middle)
distribute by hash (last_name);

INSERT INTO x1(customer_sk, customer_id, first_name) VALUES (3769, 'abcdef', 'Grace');

SELECT customer_id, octet_length(customer_id) FROM x1;
     customer_id       | octet_length
-----------------------+--------------
 abcdef                |           20
(1 row)
```

说明

两个unknown文本缺省解析成text，这样就允许||操作符解析成text连接。然后操作符的text结果转换成bpchar("空白填充的字符型"， character类型内部名称)以匹配目标字段类型。不过，从text到bpchar的转换是二进制兼容的，这样的转换是隐含的并且实际上不做任何函数调用。最后，在系统表里找到长度转换函数bpchar(bpchar, integer, boolean) 并且应用于该操作符的结果和存储的字段长。这个类型相关的函数执行所需的长度检查和额外的空白填充。

8.5 UNION，CASE 和相关构造

SQL UNION构造把不相同的数据类型进行匹配输出为统一的数据类型结果集。因为SELECT UNION语句中的所有查询结果必须在一列里显示出来，所以每个SELECT子句中的元素类型必须相互匹配并转换成一个统一的数据类型。类似地，一个CASE构造的结果表达式必须转换成统一的类型，这样整个case表达式会有一个统一的输出类型。同样的要求也存在于ARRAY构造以及GREATEST和LEAST函数中。

UNION，CASE 和相关构造解析

- 如果所有输入都是相同的数据类型，不包括unknown类型（即输入的字符串文本未声明类型，该文本首先被定义成一个未知类型），那么解析成所输入的相同数据类型。
- 如果所有输入都是unknown类型则解析成text类型（字符串类型范畴的首选类型）。否则，忽略unknown输入。
- 如果输入不属于同一个类型范畴，查询失败（unknown类型除外）。
- 如果输入类型是同一个类型范畴，则选择该类型范畴的首选类型。（例外：union操作会选择第一个分支的类型作为所选类型。）

  说明

  系统表pg_type中typcategory表示数据类型范畴， typispreferred表示是否是typcategory分类中的首选类型。
- 把所有输入转换为所选的类型（对于字符串保持原有长度）。如果从给定的输入到所选的类型没有隐式转换则失败。
- 若输入中含json、txid_snapshot、sys_refcursor或几何类型，则不能进行union。

对于 CASE、COALESCE、IF 和 IFNULL，在 TD 兼容模式下的处理

- 如果所有输入都是相同的数据类型，不包括unknown类型，那么解析成所输入的相同数据类型。
- 如果所有输入都是unknown类型则解析成text类型。
- 如果输入字符串（包括unknown，unknown当text来处理）和数字类型，那么解析成字符串类型，如果是其他不同的类型范畴，则报错。
- 如果输入类型是同一个类型范畴，则选择该类型的优先级较高的类型。
- 把所有输入转换为所选的类型。如果从给定的输入到所选的类型没有隐式转换则失败。

对于 CASE、COALESCE、IF 和 IFNULL，在 MySQL 兼容模式下的处理

- 如果所有输入都是相同的类型，不包括unknown类型，那么解析成所输入的相同数据类型。

- 如果所有输入都是unknown类型则解析成text类型。
- 如果输入是unknown类型和某一非unknown类型，则解析成该非unknown类型。
- 如果存在多种非unknown类型，将enum类型当做text类型，再进行比较。
- 如果输入类型是同一个类型范畴，则选择该类型的优先级较高的类型。如果是不同的类型范畴，则解析成text类型。
- 把所有输入转换为所选的类型。如果从给定的输入到所选的类型没有隐式转换则失败。

示例

示例1：Union中的未知类型解析。示例中未知类型文本'b'将被解析成text类型。

```
SELECT text 'a' AS "text" UNION SELECT 'b';
 text
------
 a
 b
(2 rows)
```

示例2：简单Union中的类型解析。文本1.2是numeric类型，且integer类型值1可以隐式地转换为numeric，因此使用numeric类型。

```
SELECT 1.2 AS "numeric" UNION SELECT 1;
 numeric
---------
       1
     1.2
(2 rows)
```

示例3：转换Union中的类型解析。示例中由于类型real不能被隐式地转换为integer，而integer可以隐式地转换成real类型，那么联合结果类型被系统决定为real。

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);
 real
------
    1
  2.2
(2 rows)
```

示例4：COALESCE函数输入int和varchar类型，ORA模式下会报错，TD模式下解析为varchar类型，MySQL模式下解析为text类型。

```
--指定兼容模式创建数据库ora_db、td_db、mysql_db。
CREATE DATABASE ora_db dbcompatibility = 'ORA';
CREATE DATABASE td_db dbcompatibility = 'TD';
CREATE DATABASE mysql_db dbcompatibility = 'MySQL';

--切换数据库为ora_db。
\c ora_db

--创建表t1。
ora_db=# CREATE TABLE t1(a int, b varchar(10));

--查看coalesce参数输入int和varchar类型的查询语句的执行计划。
ora_db=# EXPLAIN SELECT coalesce(a, b) FROM t1;
ERROR:  COALESCE types integer and character varying cannot be matched
CONTEXT:  referenced column: coalesce

--删除表。
ora_db=# DROP TABLE t1;

--切换数据库为td_db。
ora_db=# \c td_db
```



```
--创建表t2。
td_db=# CREATE TABLE t2(a int, b varchar(10));

--查看coalesce参数输入int和varchar类型的查询语句的执行计划。
td_db=# EXPLAIN VERBOSE SELECT coalesce(a, b) FROM t2;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 id |                 operation                  | E-rows | E-distinct | E-width | E-costs
----+--------------------------------------------+--------+------------+---------+---------
  1 | ->  Data Node Scan on "__REMOTE_FQS_QUERY__" |      0 |            |       0 | 0.00

                    Targetlist Information (identified by plan id)
-------------------------------------------------------------------------------------------------
  1 --Data Node Scan on "__REMOTE_FQS_QUERY__"
        Output: (COALESCE((t2.a)::character varying, t2.b))
        Node/s: All datanodes
        Remote query: SELECT COALESCE(a::character varying, b) AS "coalesce" FROM public.t2
(10 rows)

--删除表。
td_db=# DROP TABLE t2;

--切换数据库为mysql_db。
td_db=# \c mysql_db

--创建表t3。
mysql_db=# CREATE TABLE t3(a int, b varchar(10));

--查看coalesce参数输入int和varchar类型的查询语句的执行计划。
mysql_db=# EXPLAIN VERBOSE SELECT coalesce(a, b) FROM t3;
                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 id |                 operation                  | E-rows | E-distinct | E-width | E-costs
----+--------------------------------------------+--------+------------+---------+---------
  1 | ->  Data Node Scan on "__REMOTE_FQS_QUERY__" |      0 |            |       0 | 0.00

                    Targetlist Information (identified by plan id)
-------------------------------------------------------------------------------------------------
  1 --Data Node Scan on "__REMOTE_FQS_QUERY__"
        Output: (COALESCE((t3.a)::text, (t3.b)::text))
        Node/s: All datanodes
        Remote query: SELECT COALESCE(a::text, b::text) AS "coalesce" FROM public.t3
(10 rows)

--删除表。
mysql_db=# DROP TABLE t3;

--切换数据库为gaussdb
mysql_db=# \c gaussdb

--删除数据库。
DROP DATABASE ora_db;
DROP DATABASE td_db;
DROP DATABASE mysql_db;
```


9 全文检索

9.1 介绍

9.1.1 全文检索概述

全文检索（或者说文本搜索）提供了查询可读性文档的能力，并且通过查询相关度将结果进行排序。搜索最常见的方式是：找到包含指定查询词的所有记录，并且按照查询顺序返回这些记录。

文本搜索操作符在数据库中已存在多年。GaussDB(DWS)为文本数据类型提供~、~*、LIKE和ILIKE操作符，但这些操作符缺乏现代信息系统所要求的许多必要属性，不过这一问题可以通过使用索引及词典进行解决。

说明

实时数仓（单机部署）暂不支持全文检索功能。

文本检索缺乏信息系统所要求的必要属性：

- 没有语义支持，即使是英语也是如此。

  要识别派生词并不是那么容易，即使正则表达式也不能满足要求。例如satisfies和satisfy，当使用正则表达式寻找satisfy时，并不会查询到包含satisfies的文档。用户可以使用OR搜索多种派生形式，但过程非常繁琐。并且有些词会有上千的派生词，容易出错。
- 没有对搜索结果的分类（排序）。当搜索出成千的文档时，查找效率很低。
- 由于没有索引的支持，每一次的搜索需要遍历所有的文档，整体搜索比较缓慢。

使用全文索引可以对文档进行预处理，并且可以使后续的搜索更快速。预处理过程包括：

- 将文档解析成token。

  为每个文档标记不同类别的token是非常有必要的，例如：数字、文字、复合词、电子邮件地址，这样就可以针对不同类别做不同的处理。原则上token的类别依赖于具体的应用，但对于大多数的应用来说，可以使用一组预定义的token类。
- 将token转换为词素。

  词素像token一样是一个字符串，但它已经标准化处理，这样同一个词的不同形式是一样的。例如，标准化通常包括：将大写字母折成小写字母、删除后缀（如英

语中的s或者es）。这将允许通过搜索找到同一个词的不同形式，不需要繁琐地输入所有可能的变形样式。同时，这一步通常会删除停用词。这些停用词通常因为太常见而对搜索无用。（总之，token是文档文本的原片段，而词素被认为是有用的索引和搜索词。）GaussDB(DWS)使用词典执行这一步，且提供了各种标准的词典。

- 保存搜索优化后的预处理文档。

  比如，每个文档可以呈现为标准化词素的有序组合。伴随词素，通常还需要存储词素位置信息以用于邻近排序。因此文档包含的查询词越密集其排序越高。

词典能够对token如何标准化做到细粒度控制。使用合适的词典，可以定义不被索引的停用词。

数据类型tsvector用于存储预处理文档，tsquery用于存储查询条件，详细内容可参见**文本搜索类型**。为数据类型tsvector提供的函数和操作符可参见**文本检索函数和操作符**，其中最重要的是匹配运算符@@，请参见**基本文本匹配**。

9.1.2 文档概念

文档是全文搜索系统的搜索单元，例如：杂志上的一篇文章或电子邮件消息。文本搜索引擎必须能够解析文档，而且可以存储父文档的关联词素（关键词）。后续，这些关联词素用来搜索包含查询词的文档。

在GaussDB(DWS)中，文档通常是一个数据库表中的一行文本字段，或者这些字段的可能组合（级联）。文档可能存储在多个表中或者动态获取。换句话说，一个文档由被索引化的不同部分构成，可以不作为整体存储在任何地方。比如：

```
SELECT d_dow || '-' || d_dom || '-' || d_fy_week_seq  AS identify_serials FROM tpcds.date_dim WHERE
d_fy_week_seq = 1;
identify_serials
------------------
 5-6-1
 0-8-1
 2-3-1
 3-4-1
 4-5-1
 1-2-1
 6-7-1
(7 rows)
```

须知

实际上，在这些示例查询中，应该使用coalesce防止一个独立的NULL属性导致整个文档的NULL结果。

另外一种可能是：文档在文件系统中作为简单的文本文件存储。在这种情况下，数据库可以用于存储全文索引并且执行搜索，同时可以使用一些唯一标识从文件系统中检索文档。然而，从数据库外部检索文件需要拥有系统管理员权限或者特殊函数支持。因此，还是将所有数据保存在数据库中比较方便。同时，将所有数据保存在数据库中可以方便地访问文档元数据以便于索引和显示。

为了实现文本搜索目的，必须将每个文档减少至预处理后的tsvector格式。搜索和相关性排序都是在tsvector形式的文档上执行的。原始文档只有在被选中要呈现给用户时才会被当检索。因此，常将tsvector说成文档，但是很显然其实它只是完整文档的一种紧凑表示。

9.1.3 基本文本匹配

GaussDB(DWS)的全文检索基于匹配算子@@，当一个tsvector(document)匹配到一个tsquery(query)时，则返回true。其中，tsvector(document)和tsquery(query)两种数据类型可以任意排序。

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery AS RESULT;
 result
----------
 t
(1 row)
SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector AS RESULT;
 result
----------
 f
(1 row)
```

正如上面例子表明，tsquery不仅是文本，且比tsvector包含的要多。tsquery包含已经标注化为词条的搜索词，同时可能是使用AND、OR、或NOT操作符连接的多个术语。详细请参见**文本搜索类型**。函数to_tsquery和plainto_tsquery对于将用户书写文本转换成适合的tsquery是非常有用的，比如将文本中的词标准化。类似地，to_tsvector用于解析和标准化文档字符串。因此，实际中文本搜索匹配看起来更像这样：

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat') AS RESULT;
result
----------
 t
(1 row)
```

需要注意的是，下面这种方式是不可行的：

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat')AS RESULT;
result
----------
 f
(1 row)
```

由于tsvector没有对rats进行标准化，所以rats不匹配rat。

@@操作符也支持text输入，允许一个文本字符串的显示转换为tsvector或者在简单情况下忽略tsquery。可用形式是：

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

形式text @@ tsquery等价于to_tsvector(text) @@ tsquery，而text @@ text等价于to_tsvector(text) @@ plainto_tsquery(text)。

9.1.4 分词器

全文检索功能还可以做更多事情：忽略索引某个词（停用词），处理同义词和使用复杂解析，例如，不仅基于空格的解析。这些功能通过文本搜索分词器控制。GaussDB(DWS)支持多语言的预定义的分词器，并且可以创建分词器（gsql的\dF命令显示了所有可用分词器）。

在安装期间选择一个合适的分词器，并且在postgresql.conf中相应的设置default_text_search_config。如果为了整个集群使用同一个文本搜索分词器可以使用postgresql.conf中的值。如果需要在集群中使用不同分词器，可以使用ALTER DATABASE ... SET在任一数据库进行配置。用户也可以在每个会话中设置default_text_search_config。

每个依赖于分词器的文本搜索函数有一个可选的配置参数，用以明确声明所使用的分词器。仅当忽略这个参数的时候，才使用default_text_search_config。

为了更方便的建立自定义文本搜索分词器，可以通过简单的数据库对象建立分词器。GaussDB(DWS)文本搜索功能提供了四种类型与分词器相关的数据库对象：

- 文本搜索解析器将文档分解为token，并且分类每个token（例如：词和数字）。
- 文本搜索词典将token转换成规范格式并且丢弃停用词。
- 文本搜索模板提供潜在的词典功能：一个词典指定一个模板，并且为模板设置参数。
- 文本搜索分词器选择一个解析器，并且使用一系列词典规范化语法分析器产生的token。

9.1.5 限制约束

GaussDB(DWS)的全文检索功能当前限制约束是：

- 每个分词长度必须小于2K字节。
- tsvector结构（分词+位置）的长度必须小于1兆字节。
- tsvector的位置值必须大于0，小于等于16,383。
- 每个分词在文档中位置数必须小于256，若超过将舍弃后面的位置信息。
- tsquery中的关键字及对应运算符最大支持到32,768。

9.2 在数据库表中搜索文本

9.2.1 搜索表

本章节主要介绍如何使用文本搜索运算符搜索数据库表。

- 一个简单查询：将body字段中包含science的每一行打印出来。

```
DROP SCHEMA IF EXISTS tsearch CASCADE;

CREATE SCHEMA tsearch;

CREATE TABLE tsearch.pgweb(id int, body text, title text, last_mod_date date);

INSERT INTO tsearch.pgweb VALUES(1, 'Philology is the study of words, especially the history and development of the words in a particular language or group of languages.', 'Philology', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(2, 'Mathematics is the science that deals with the logic of shape, quantity and arrangement.', 'Mathematics', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(3, 'Computer science is the study of processes that interact with data and that can be represented as data in the form of programs.', 'Computer science', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(4, 'Chemistry is the scientific discipline involved with elements and compounds composed of atoms, molecules and ions.', 'Chemistry', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(5, 'Geography is a field of science devoted to the study of the lands, features, inhabitants, and phenomena of the Earth and planets.', 'Geography', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(6, 'History is a subject studied in schools, colleges, and universities that deals with events that have happened in the past.', 'History', '2010-1-1');

INSERT INTO tsearch.pgweb VALUES(7, 'Medical science is the science of dealing with the maintenance of health and the prevention and treatment of disease.', 'Medical science', '2010-1-1');
```

```
INSERT INTO tsearch.pgweb VALUES(8, 'Physics is one of the most fundamental scientific disciplines,
and its main goal is to understand how the universe behaves.', 'Physics', '2010-1-1');


SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector('english', body) @@
to_tsquery('english', 'science');
 id |                                                 body                                                 |  title
----
+--------------------------------------------------------------------------------------------------------------------------
----+---------

 2 | Mathematics is the science that deals with the logic of shape, quantity and
arrangement.                                        | Mathematics
 3 | Computer science is the study of processes that interact with data and that can be represented as
data in the form of programs. | Computer science
 5 | Geography is a field of science devoted to the study of the lands, features, inhabitants, and
phenomena of the Earth and planets.   | Geography
 7 | Medical science is the science of dealing with the maintenance of health and the prevention and
treatment of disease.           | Medical science
(4 rows)
```

像science这样的相关词都会被找到，因为这些词都被处理成了相同标准的词条。

上面的查询指定english配置来解析和规范化字符串。也可以省略此配置，通过default_text_search_config进行配置设置：

```
SHOW default_text_search_config;
 default_text_search_config
----------------------------
 pg_catalog.english
(1 row)

SELECT id, body, title FROM tsearch.pgweb WHERE to_tsvector(body) @@ to_tsquery('science');
 id |                                                 body                                                 |  title
----
+--------------------------------------------------------------------------------------------------------------------------
----+---------

 2 | Mathematics is the science that deals with the logic of shape, quantity and
arrangement.                                        | Mathematics
 3 | Computer science is the study of processes that interact with data and that can be represented as
data in the form of programs. | Computer science
 5 | Geography is a field of science devoted to the study of the lands, features, inhabitants, and
phenomena of the Earth and planets.   | Geography
 7 | Medical science is the science of dealing with the maintenance of health and the prevention and
treatment of disease.           | Medical science

(4 rows)
```

- 一个复杂查询：检索出在title或者body字段中包含treatment和science的最近10篇文档：

```
SELECT title FROM tsearch.pgweb WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('treatment &
science') ORDER BY last_mod_date DESC LIMIT 10;
 title
--------
Medical science
(1 rows)
```

为了清晰，举例中没有调用coalesce函数在两个字段中查找包含NULL的行。

以上例子均在没有索引的情况下进行查询。对于大多数应用程序来说，这个方法很慢。因此除了偶尔的特定搜索，文本搜索在实际使用中通常需要创建索引。

9.2.2 创建 GIN 索引

为了加速文本搜索，可以创建GIN索引。

```
CREATE INDEX pgweb_idx_1 ON tsearch.pgweb USING gin(to_tsvector('english', body));
```

to_tsvector()函数有两个版本，只输一个参数的版本和输两个参数的版本。

只输一个参数时，系统默认采用default_text_search_config所指定的分词器。

创建GIN索引时必须使用to_tsvector的两参数版本，否则索引内容可能不一致。只有指定了分词器名称的全文检索函数才可以在索引表达式中使用。因为索引的内容必须不受default_text_search_config的影响。由于default_text_search_config的值可以随时调整，从而导致不同条目生成的tsvector采用了不同的分词器，并且无法区分究竟使用了哪个分词器。正确地转储和恢复这样的索引也是不支持的。

在上述创建索引中to_tsvector使用了两个参数，只有当查询时也使用了两个参数，且参数值与索引中相同时，才会使用该索引。例如WHERE to_tsvector('english', body) @@ 'a & b' 可以使用索引，但WHERE to_tsvector(body) @@ 'a & b'不能使用索引。这可确保索引各条目是使用相同的分词器创建的。

索引中的分词器名称由另一列指定时可以建立更复杂的表达式索引。例如：

```
CREATE INDEX pgweb_idx_2 ON tsearch.pgweb USING gin(to_tsvector('zhparser', body));
```

说明

本示例中zhparser仅支持UTF8/GBK的数据库编码格式，在Encoding为SQL_ASCII下会报错。

其中body是pgweb表中的一列。当对索引的各条目使用了哪个分词器进行记录时，允许在同一索引中存在混合分词器。当文档集合中包含不同语言的文档时，这将是有用的。再次强调，打算使用索引的查询必须措辞匹配，例如，WHERE to_tsvector(config_name, body) @@ 'a & b'与索引中的to_tsvector措辞匹配。

索引甚至可以连接列：

```
CREATE INDEX pgweb_idx_3 ON tsearch.pgweb USING gin(to_tsvector('english', title || ' ' || body));
```

另一个方法是创建一个单独的tsvector列控制to_tsvector的输出。下面的例子是title和body的连接， 当其它是NULL的时候，使用coalesce确保一个字段仍然会被索引：

```
ALTER TABLE tsearch.pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE tsearch.pgweb SET textsearchable_index_col = to_tsvector('english', coalesce(title,'') || ' ' ||
coalesce(body,''));
```

然后为加速搜索创建一个GIN索引：

```
CREATE INDEX textsearch_idx_4 ON tsearch.pgweb USING gin(textsearchable_index_col);
```

现在，就可以执行一个快速全文搜索了：

```
SELECT title
FROM tsearch.pgweb
WHERE textsearchable_index_col @@ to_tsquery('science & Computer')
ORDER BY last_mod_date DESC
LIMIT 10;

 title
--------
 Computer science
(1 rows)
```

相比于一个表达式索引，单独列方法的一个优势是：不必在查询时显式指定分词器以便能使用索引。正如上面例子所示，查询可以依赖于default_text_search_config。另一个优势是搜索比较快速，因为它没有必要重新利用to_tsvector调用来验证索引匹配。表达式索引方法更容易建立，且它需要较少的磁盘空间，因为tsvector形式没有明确存储。

9.2.3 索引使用约束

下面是一个使用索引的例子，由于SQL_ASCII的数据库编码格式不支持中文字符，请在Encoding为UTF8/GBK的数据库中执行以下示例：

```
CREATE TBALE table1 (c_int int,c_bigint bigint,c_varchar varchar,c_text text) with(orientation=row);

CREATE TEXT SEARCH CONFIGURATION ts_conf_1(parser=POUND);
CREATE TEXT SEARCH CONFIGURATION ts_conf_2(parser=POUND) with(split_flag='%');

SET default_text_search_config='ts_conf_1';
CREATE INDEX idx1 ON table1 using gin(to_tsvector(c_text));

SET default_text_search_config='ts_conf_2';
CREATE INDEX idx2 ON table1 using gin(to_tsvector(c_text));

SELECT c_varchar,to_tsvector(c_varchar) FROM table1 where to_tsvector(c_text) @@ plainto_tsquery('
￥#@……&**')  and to_tsvector(c_text) @@ plainto_tsquery('某公司 ')  and c_varchar is not null order by 1
desc limit 3;
```

该例子的关键点是表table1的同一个列c_text上建立了两个gin索引：idx1和idx2，但这两个索引是在不同default_text_search_config的设置下建立的。该例子和同一张表的同一个列上建立普通索引的不同之处在于：

- GIN索引使用了不同的parser（即分隔符不同），那么idx1和idx2的索引数据是不同的；
- 在同一张表的同一个列上建立的多个普通索引的索引数据是相同的；

因此当执行同一个查询时，使用idx1和idx2查询出的结果是不同的。

使用约束

通过上面的例子，GIN索引使用满足如下条件时：

- 在同一个表的同一个列上建立了多个GIN索引；
- 这些GIN索引使用了不同的parser（即分隔符不同）；
- 在查询中使用了该列，且执行计划中使用索引进行扫描。

为了避免使用不同gin索引导致查询结果不同的问题，需要保证在物理表的一列上只有一个gin索引可用。

9.3 控制文本搜索

9.3.1 解析文档

GaussDB(DWS)中提供了to_tsvector函数把文档处理成tsvector数据类型。

to_tsvector([*config* **regconfig**,] *document* **text**) **returns tsvector**

to_tsvector将文本文档解析为token，再将token简化到词素，并返回一个tsvector。其中tsvector中列出了词素及它们在文档中的位置。文档是根据指定的或默认的文本搜索分词器进行处理的。这里有一个简单的例子：

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

通过以上例子可发现结果tsvector不包含词a、on或者it，rats变成rat，并且忽略标点符号-。

to_tsvector函数内部调用一个解析器，将文档的文本分解成token并给每个token指定一个类型。对于每个token，有一系列词典可供查询。词典系列因token类型的不同而不同。识别token的第一本词典将发出一个或多个标准词素来表示token。例如：

- rats变成rat因为词典认为词rats是rat的复数形式。
- 有些词被作为停用词（请参考**停用词**），这样它们就会被忽略，因为它们出现得太过频繁以致于搜索中没有用处。比如示例中的a、on和it。
- 如果没有词典识别token，那么它也被忽略。在上述示例中，符号“-”被忽略，因为词典没有给它分配token类型（空间符号），即空间记号永远不会被索引。

语法解析器、词典和要索引的token类型由选定的文本搜索分词器决定。可以在同一个数据库中有多种不同的分词器，以及提供各种语言的预定义分词器。在以上例子中，使用缺省分词器english。

函数setweight可以给tsvector的记录加权重，权重是字母A、B、C、D之一。这通常用于标记来自文档不同部分的记录，比如标题、正文。之后，这些信息可以用于排序搜索结果。

因为to_tsvector(NULL)会返回空，当字段可能是空的时候，建议使用coalesce。以下是为结构化文档创建tsvector的方法：

```
CREATE TABLE tsearch.tt (id int, title text, keyword text, abstract text, body text, ti tsvector);

INSERT INTO tsearch.tt(id, title, keyword, abstract, body) VALUES (1, 'book', 'literature', 'Ancient
poetry','Tang poem Song jambic verse');

UPDATE tsearch.tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
DROP TABLE tsearch.tt;
```

上例使用setweight标记已完成的tsvector中的每个词的来源，并且使用tsvector连接操作符“||”合并标记过的tsvector值，**处理tsvector**一节详细介绍了这些操作。

9.3.2 解析查询

GaussDB(DWS)提供了函数to_tsquery和plainto_tsquery将查询转换为tsquery数据类型，to_tsquery提供比plainto_tsquery更多的功能，但对其输入要求更严格。

to_tsquery

to_tsquery将查询转换为tsquery数据类型。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to_tsquery从querytext中创建一个tsquery，querytext必须由布尔运算符& (AND)、| (OR)和! (NOT)分割的单个token组成。这些运算符可以用圆括弧分组。也就是说，to_tsquery输入必须遵循tsquery输入的通用规则，具体请参见**文本搜索类型**。不同的是基本tsquery以token表面值作为输入，而to_tsquery使用指定或默认分词器将每个token标准化成词素，并依据分词器丢弃属于停用词的token。例如：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
```

```
 'fat' & 'rat'
(1 row)
```

像在基本tsquery中的输入一样，权重可以附加到每个词素来限制它只匹配那些有相同weight(s)的tsvector词素。比如：

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
(1 row)
```

同时，符号“*”也可以附加到词素来指定前缀匹配：

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
(1 row)
```

这样的词素将匹配tsquery中指定字符串和权重的项。

plainto_tsquery

plainto_tsquery将未格式化的文本querytext变换为tsquery。类似于to_tsvector，文本被解析并且标准化，然后在存在的词之间插入&(AND)布尔算子。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

比如：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
(1 row)
```

请注意，plainto_tsquery无法识别布尔运算符、权重标签，或在其输入中的前缀匹配标签：

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
(1 row)
```

在这里，所有输入的标点符号作为空格符号丢弃。

9.3.3 排序查询结果

排序是指试图针对特定查询衡量文档的相关度，从而将众多的匹配文档中相关度最高的文档排在最前。GaussDB(DWS)提供了两个预置的排序函数:ts_rank和ts_rank_cd。函数考虑了词法，距离，和结构信息；也就是，考虑查询词在文档中出现的频率、紧密程度、以及出现的地方在文档中的重要性。然而，相关性的概念是模糊的，并且是跟应用强相关的。不同的应用程序可能需要额外的信息来排序，比如，文档的修改时间，内置的排序函数等。也可以自定义排序函数或者采用附加因素组合这些排序函数的结果来满足特定需求。

两个预置的排序函数：

```
ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4
```

基于词素匹配率对vector进行排序：

```
ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization integer ]) returns float4
```

该函数需要位置信息的输入。因此它不能在"剥离"tsvector值的情况下运行—它将总是返回零。

对于这两个函数，可选的权重参数提供给词加权重的能力，词的权重大小取决于所加的权值。权重阵列指定在排序时为每类词汇加多大的权重。

```
{D-weight, C-weight, B-weight, A-weight}
```

如果没有提供权重，则使用缺省值：{0.1, 0.2, 0.4, 1.0}

通常的权重是用来标记文档特殊领域的词，如标题或最初的摘要，所以相对于文章主体中的词它们有着更高或更低的重要性。

由于较长的文档有更多的机会包含查询词，因此有必要考虑文档的大小。例如，包含有5个搜索词的一百字文档比包含有5个搜索词的一千字文档相关性更高。两个预置的排序函数都采用了一个整型的标准化选项来定义文档长度是否影响排序及如何影响。这个整型选项控制多个行为，所以它是一个屏蔽字：可以使用“|”指定一个或多个行为（例如，2|4）。

- 0（缺省）表示：跟长度大小没有关系
- 1 表示：排名（rank）除以(文档长度的对数+1)
- 2表示：排名除以文档的长度
- 4表示：排名除以两个扩展词间的调和平均距离。只能使用ts_rank_cd实现
- 8表示：排名除以文档中单独词的数量
- 16表示：排名除以单独词数量的对数+1
- 32表示：排名除以排名本身+1

当指定多个标志位时，会按照所列的顺序依次进行转换。

需要特别注意的是，排序函数不使用任何全局信息，所以不可能产生一个某些情况下需要的1%或100%的理想标准值。标准化选项32 (rank/(rank+1))可用于所有规模的从零到一之间的排序。而这只是一个表面变化，并不会影响搜索结果的排序。

下面是一个例子，仅选择排名前十的匹配：

由于SQL_ASCII的数据库编码格式不支持中文字符，请在Encoding为UTF8/GBK的数据库中执行以下示例：

```
SELECT id, title, ts_rank_cd(to_tsvector(body), query) AS rank
FROM tsearch.pgweb, to_tsquery('science') query
WHERE query @@ to_tsvector(body)
ORDER BY rank DESC
LIMIT 10;
 id |      title       | rank
----+------------------+------
  7 | Medical science  |   .2
  3 | Computer science |   .1
  2 | Mathematics      |   .1
  5 | Geography        |   .1
(4 rows)
```

这是使用标准化排序的相同例子：

```
SELECT id, title, ts_rank_cd(to_tsvector(body), query, 32 /* rank/(rank+1) */ ) AS rank
FROM tsearch.pgweb, to_tsquery('science') query
WHERE  query @@ to_tsvector(body)
ORDER BY rank DESC
LIMIT 10;
 id |      title       |   rank
----+------------------+----------
  7 | Medical science  | .166667
```



```
 3 | Computer science | .0909091
 2 | Mathematics      | .0909091
 5 | Geography        | .0909091
(4 rows)
```

下面是使用中文分词法排序查询的例子：

```
CREATE TABLE tsearch.ts_zhparser(id int, body text);
INSERT INTO tsearch.ts_zhparser VALUES(1, '排序');
INSERT INTO tsearch.ts_zhparser VALUES(2, '排序查询');
INSERT INTO tsearch.ts_zhparser VALUES(3, '查询排序');
--精确匹配
SELECT id, body, ts_rank_cd(to_tsvector('zhparser',body), query) AS rank FROM tsearch.ts_zhparser,
to_tsquery('排序') query WHERE query @@ to_tsvector(body);
 id | body | rank
----+------+------
  1 | 排序 |   .1
(1 row)

--模糊匹配
SELECT id, body, ts_rank_cd(to_tsvector('zhparser',body), query) AS rank FROM tsearch.ts_zhparser,
to_tsquery('排序') query WHERE query @@ to_tsvector('zhparser',body);
 id |   body   | rank
----+----------+------
  3 | 查询排序 |   .1
  1 | 排序     |   .1
  2 | 排序查询 |   .1
(3 rows)
```

排序要遍历每个匹配的tsvector，因此资源消耗多，可能会因为I/O限制导致排序慢。可是这是很难避免的，因为实际查询中通常会有大量的匹配。

9.3.4 高亮搜索结果

搜索结果的理想显示是列出每篇文档中与搜索相关的部分，并标识为什么与查询相关。搜索引擎能够显示标识了搜索词的文档片段。GaussDB(DWS)提供了函数ts_headline支持这部分功能。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

ts_headline的输入是带有查询条件的文档，其返回文档中的摘录，在摘录中查询词是高亮显示的。用来解析文档的分词器由config参数指定。如果省略config，则使用default_text_search_config的值所指定的分词器。

指定options字符串时，需由一个或多个option=value对组成，且必须用逗号分隔。options可以是下面的选项：

- StartSel，StopSel：分隔文档中出现的查询词，以区别于其他摘录词。当包含有空格或逗号时，必须用双引号将字符串引起来。
- MaxWords，MinWords：定义摘录的最长和最短值。
- ShortWord：在摘录的开始和结束会丢弃此长度或更短的词。默认值为3会消除常见的英语冠词。
- HighlightAll：布尔标志。如果为true，则整个文档将作为摘录，忽略前面三个参数的值。
- MaxFragments：要显示的文本摘录或片段的最大数量。默认值0表示选择非片段的摘录生成方法。 大于0的值表示选择基于片段的摘录生成。此方法查找带有尽可能多查询词的文本片段，并显示查询词周围的上下文片段。因此，查询词临近每个片段的中间，且查询词两边都有词。每个片段至多有MaxWords，并且在每一个片段开始和结束删除长度为ShortWord或更短的词。如果在文档中没有找到所有的查询词，则文档中开头将显示MinWords单片段。

- FragmentDelimiter：当有一个以上的片段时，通过该字符串分隔这些片段。

不声明选项时，采用下面的缺省值：

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

比如：

```
SELECT ts_headline('english','The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.',to_tsquery('english', 'query & similarity'));
                                        ts_headline
-----------------------------------------------------------------------------------------------------------------
 containing given <b>query</b> terms and return them in order of their <b>similarity</b> to the <b>query</b>.
(1 row)

SELECT ts_headline('english','The most common type of search is to find all documents containing given query terms and return them in order of their similarity to the query.',to_tsquery('english', 'query & similarity'),'StartSel = <, StopSel = >');
                                      ts_headline
-------------------------------------------------------------------------------------------------
 containing given <query> terms and return them in order of their <similarity> to the <query>.
(1 row)
```

ts_headline使用原始文档，而不是tsvector摘录，因此使用起来会慢，应慎重使用。

9.4 附加功能

9.4.1 处理 tsvector

GaussDB(DWS)提供了用来操作tsvector类型的函数和操作符。

- tsvector || tsvector

  tsvector连接操作符返回一个新的tsvector类型，它综合了两个tsvector中词素和位置信息，并保留词素的位置信息和权重标签。右侧的tsvector的起始位置位于左侧tsvector的最后位置，因此，新生成的tsvector几乎等同于将两个原始文档字串连接后进行to_tsvector操作。（这个等价是不准确的，因为任何从左边tsvector中删除的停用词都不会影响结果，但是，在使用文本连接时，则会影响词素在右侧tsvector中的位置。）

  相较于对文本进行连接后再执行to_tsvector操作，使用tsvector类型进行连接操作的优势在于可以对文档的不同部分使用不同配置进行解析。因为setweight函数会对给定的tsvector中的语素进行统一设置，如果想要对文档的不同部分设置不同的权重，需要在连接之前对文本进行解析和权重设置。

- setweight(vector tsvector, weight "char") returns tsvector

  setweight返回一个输入tsvector的副本，其中每一个位置都使用给定的权重做了标记。权值可以为A、B、C或D（D是tsvector副本的默认权重，并且不在输出中呈现）。当对tsvector进行连接操作时，这些权重标签将会被保留，文档不同部分以不同的权重进行排序。

须知

权重标签作用于位置，而不是词素。如果传入的tsvector已经被剥离了位置信息，那么setweight函数将什么都不做。

- length(vector tsvector) returns integer

  返回vector中的词素的数量。

- strip(vector tsvector) returns tsvector

  返回一个tsvector类型，其中包含输入的tsvector的同义词，但不包含任何位置和权重信息。虽然在相关性排序中，这里返回的tsvector要比未拆分的tsvector的作用小很多，但它通常都比未拆分的tsvector小的多。

9.4.2 处理 tsquery

GaussDB(DWS)提供了函数和操作符用来操作tsquery类型的查询。

- tsquery && tsquery

  返回两个给定查询tsquery的与结果。

- tsquery || tsquery

  返回两个给定查询tsquery的或结果。

- !! tsquery

  返回给定查询tsquery的非结果。

- numnode(query tsquery) returns integer

  返回tsquery中的节点数目（词素加操作符），这个函数在检查查询是否有效（返回值大于0），或者只包含停用词（返回值等于0）时，是有用的。例如：

  ```
  SELECT numnode(plainto_tsquery('the any'));
  NOTICE:  text-search query contains only stop words or doesn't contain lexemes, ignored
  CONTEXT:  referenced column: numnode
   numnode
  ---------
        0
  (1 row)

  SELECT numnode('foo & bar'::tsquery);
   numnode
  ---------
        3
  (1 row)
  ```

- querytree(query tsquery) returns text

  返回可用于索引搜索的tsquery部分，该函数对于检测非索引查询是有用的（例如只包含停用词或否定项）。例如:

  ```
  SELECT querytree(to_tsquery('!defined'));
   querytree
  ------------
   T
  (1 row)
  ```

9.4.3 查询重写

ts_rewrite函数可以从tsquery中搜索一个特定的目标子查询，并在该子查询每次出现的地方都替换为另一个子查询。 实际上这只是通过字串替换而得到的一个特定tsquery版本。目标子查询和替换查询组合起来可以被认为是一个重写规则。一组类似的重写规则可以为搜索提供强大的帮助。例如，可以使用同义词扩大搜索范围（例如new york，big apple，nyc，gotham）或限制搜索范围在用户直接感兴趣的热点话题上。

- ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery

  ts_rewrite的这种形式只适用于一个单一的重写规则：任何出现目标子查询的地方都被无条件替换。例如：


```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
(1 row)
```

- ts_rewrite (query tsquery, select text) returns tsquery

  ts_rewrite的这种形式接受一个起始查询和SQL查询命令。 这里的查询命令是文本字串形式，必须产生两个tsquery列。查询结果的每一行，第一个字段的值（目标子查询） 都会被第二个字段（替代子查询）替换。

  说明

  当多个规则需要重写时，重写顺序非常重要； 因此在实践中需要使用ORDER BY将源查询按照某些字段进行排序。

  例如：举一个现实生活中天文学上的例子。将使用表驱动的重写规则扩大supernovae的查询范围：

```
CREATE TABLE tsearch.aliases (id int, t tsquery, s tsquery);

INSERT INTO tsearch.aliases VALUES(1, to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM tsearch.aliases');
          ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
(1 row)
```

  可以通过更新表修改重写规则：

```
UPDATE tsearch.aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT t, s FROM tsearch.aliases');
                ts_rewrite
-----------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
(1 row)
```

  需要重写的规则越多，重写操作就越慢。因为它要检查每一个可能匹配的规则。为了过滤明显的非候选规则，可以使用tsquery类型的操作符来实现。在下面的例子中， 只选择那些可能与原始查询匹配的规则：

```
SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM tsearch.aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'a'
(1 row)
DROP TABLE ts_rewrite;
```

9.4.4 收集文档统计信息

函数ts_stat可用于检查配置和查找候选停用词。

```
ts_stat(sqlquery text, [ weights text, ]
      OUT word text, OUT ndoc integer,
      OUT nentry integer) returns setof record
```

sqlquery是一个包含SQL查询语句的文本，该SQL查询将返回一个tsvector。ts_stat执行SQL查询语句并返回一个包含tsvector中每一个不同的语素（词）的统计信息。返回信息包括：

- word text：词素。
- ndoc integer：词素在文档（tsvector）中的编号。

- nentry integer：词素出现的频率。

如果设置了权重条件，只有标记了对应权重的词素才会统计频率。例如，在一个文档集中检索使用频率最高的十个单词：

```
SELECT * FROM ts_stat('SELECT to_tsvector(''english'', sr_reason_sk) FROM tpcds.store_returns WHERE sr_customer_sk < 10') ORDER BY nentry DESC, ndoc DESC, word LIMIT 10;
  word | ndoc | nentry
------+------+--------
 32   |    2 |      2
 33   |    2 |      2
 1    |    1 |      1
 10   |    1 |      1
 13   |    1 |      1
 14   |    1 |      1
 15   |    1 |      1
 17   |    1 |      1
 20   |    1 |      1
 22   |    1 |      1
(10 rows)
```

同样的情况，但是只计算权重为A或者B的单词使用频率：

```
SELECT * FROM ts_stat('SELECT to_tsvector(''english'', sr_reason_sk) FROM tpcds.store_returns WHERE sr_customer_sk < 10', 'a') ORDER BY nentry DESC, ndoc DESC, word LIMIT 10;
 word | ndoc | nentry
------+------+--------
(0 rows)
```

9.5 文本搜索解析器

文本搜索解析器负责将原文档文本分解为多个token，并标识每个token的类型。这里的类型集由解析器本身定义。注意，解析器并不修改文本，它只是确定合理的单词边界。由于这一限制，人们更需要定制词典，而不是为每个应用程序定制解析器。

目前GaussDB(DWS)提供了四个内置的解析器，分别为pg_catalog.default/pg_catalog.ngram/pg_catalog.zhparser/pg_catalog.pound，其中pg_catalog.default适用于英文分词场景，pg_catalog.ngram/pg_catalog.zhparser/pg_catalog.pound是为了支持中文全文检索功能新增的三种解析器，适用于中文及中英混合分词场景。

内置解析器pg_catalog.default，它能识别23种token类型，显示在表9-1中。

表 9-1 默认解析器类型

别名	描述	示例
asciiword	Word, all ASCII letters	elephant
word	Word, all letters	mañana
numword	Word, letters and digits	beta1
asciihword	Hyphenated word, all ASCII	up-to-date
hword	Hyphenated word, all letters	lógico-matemática
numhword	Hyphenated word, letters and digits	postgresql-beta1

别名	描述	示例
hword_asciipart	Hyphenated word part, all ASCII	postgresql in the context postgresql-beta1
hword_part	Hyphenated word part, all letters	lógico or matemática in the context lógico-matemática
hword_numpart	Hyphenated word part, letters and digits	beta1 in the context postgresql-beta1
email	Email address	foo@example.com
protocol	Protocol head	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	URL path	/stuff/index.html, in the context of a URL
file	File or path name	/usr/local/foo.txt, if not within a URL
sfloat	Scientific notation	-1.23E+56
float	Decimal notation	-1.234
int	Signed integer	-1234
uint	Unsigned integer	1234
version	Version number	8.3.0
tag	XML tag	<a href="dictionaries.html">
entity	XML entity	&
blank	Space symbols	(any whitespace or punctuation not otherwise recognized)

注意：对于解析器来说，一个“字母”的概念是由数据库的语言区域设置，即lc_ctype设置决定的。只包含基本ASCII字母的词被报告为一个单独的token类型，因为这类词有时需要被区分出来。大多数欧洲语言中，对token类型word和asciiword的处理方法是类似的。

email不支持某些由RFC 5322定义的有效电子邮件字符。具体来说，可用于email用户名的非字母数字字符仅包含句号、破折号和下划线。

解析器可能对同一内容进行重叠token。例如，包含连字符的单词将作为一个整体进行报告，其组件也会分别被报告：

```
SELECT alias, description, token FROM ts_debug('english','foo-bar-beta1');
     alias      |              description              |     token
----------------+---------------------------------------+---------------
 numhword       | Hyphenated word, letters and digits   | foo-bar-beta1
```

```
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
(6 rows)
```

这种行为是有必要的，因为它支持搜索整个复合词和各组件。这里是另一个例子：

```
SELECT alias, description, token FROM ts_debug('english','http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
(4 rows)
```

N-gram是一种机械分词方法，适用于无语义中文分词场景。N-gram支持中文编码包括GBK、UTF-8。内置6种token类型，如表9-2 所示。

表 9-2 token 类型

Alias	Description
zh_words	chinese words
en_word	english word
numeric	numeric data
alnum	alnum string
grapsymbol	graphic symbol
multisymbol	multiple symbol

Zhparser是基于词典的语义分词方法，底层调用SCWS(https://github.com/hightman/scws)分词算法，适用于有语义的中文分词场景。SCWS是一套基于词频词典的机械式中文分词引擎，可以将一整段的中文文本正确地切分成词。支持GBK、UTF-8两种中文编码格式。内置26种token类型如表9-3所示：

表 9-3 token 类型

Alias	Description
A	形容词
B	区别词
C	连词
D	副词
E	叹词
F	方位词
G	语素

Alias	Description
H	前接成分
I	成语
J	简称略语
K	后接成分
L	习用语
M	数词
N	名词
O	拟声词
P	介词
Q	量词
R	代词
S	处所词
T	时间词
U	助词
V	动词
W	标点符号
X	未知词
Y	语气词
Z	状态词

Pound是一种固定格式分词方法，适用于无语意但待解析文本以固定分隔符分割开来的中英文分词场景。支持中文编码包括GBK、UTF8，支持英文编码包括ASCII。内置6种token类型，如**表4 token类型**所示；支持5种分隔符，如**表9-5**所示，在用户不进行自定义设置的情况下分隔符默认为“#”。Pound限制单个token长度不能超过256个字符。

表 9-4 token 类型

Alias	Description
zh_words	chinese words
en_word	english word
numeric	numeric data
alnum	alnum string

Alias	Description
grapsymbol	graphic symbol
multisymbol	multiple symbol

表 9-5 分隔符类型

分隔符	描述
@	Special character
#	Special character
$	Special character
%	Special character
/	Special character

9.6 词典

9.6.1 词典概述

词典用于定义停用词（stop words），即全文检索时不搜索哪些词。

词典还可以用于对同一词的不同形式进行规范化，这样同一个词的不同派生形式都可以进行匹配。规范化后的词称为词位（lexeme）。

除了提高检索质量外，词的规范化和删除停用词可以减少文档tsvector格式的大小，从而提高性能。词的规范化和删除停用词并不总是具有语言学意义，用户可以根据应用环境在词典定义文件中自定义规范化和删除规则。

一个词典是一个程序，接收标记（token）作为输入，并返回：

- 如果token在词典中已知，返回对应lexeme数组（注意，一个标记可能对应多个lexeme）。
- 一个lexeme。一个新token会代替输入token被传递给后继词典（当前词典可被称为过滤词典）。
- 如果token在词典中已知，但它是一个停用词，返回空数组。
- 如果词典不能识别输入的token，返回NULL。

GaussDB(DWS)提供了多种语言的预定义词典，同时提供了五种预定义的词典模板，分别是Simple，Synonym，Thesaurus，Ispell，和Snowball，可用于创建自定义参数的新词典。

在使用全文检索时，建议用户：

- 可以在文本搜索配置中定义一个解析器，以及一组用于处理该解析器的输出标记词典。对于解析器返回的每个标记类型，可以在配置中指定不同的词典列表进行处理。当解析器输出一种类型的标记后，在对应列表的每个词典中会查阅该标

记，直到某个词典识别它。如果它被识别为一个停用词， 或者没有任何词典识别，该token将被丢弃，即不被索引或检索到。通常情况下，第一个返回非空结果的词典决定了最终结果，后继词典将不会继续处理。但是一个过滤类型的词典可以依据规则替换输入token，然后将替换后的token传递给后继词典进行处理。

- 配置词典列表的一般规则是，第一个位置放置一个应用范围最小的、最具体化定义的词典，其次是更一般化定义的词典， 最后是一个普适定义的词典，比如Snowball词干词典或Simple词典。在下面例子中，对于一个针对天文学的文本搜索配置astro_en，可以定义标记类型asciiword（ASCII词）对应的词典列表为：天文术语的Synonym同义词词典， Ispell英语词典和Snowball英语词干词典。

  ```
  ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astro_syn, english_ispell, english_stem;
  ```

  过滤类型的词典可以放置在词典列表中除去末尾的任何地方，放置在末尾时是无效的。使用这些词典对标记进行部分规范化，可以有效简化后继词典的处理。

9.6.2 停用词

停用词是很常见的词，几乎出现在每一个文档中，并且没有区分值。因此，在全文搜索的语境下可忽视它们。停用词处理逻辑和词典类型相关。其中，Ispell词典会先对标记进行规范化，然后再查看停用词表，而Snowball词典会最先检查输入标记是否为停用词。

例如，每个英文文本包含像a和the的单词，因此没必要将它们存储在索引中。然而，停用词影响tsvector中的位置，同时位置也会影响相关度：

```
SELECT to_tsvector('english','in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

位置1、2、4是停用词，所以不显示。为包含和不包含停用词的文档计算出的排序是完全不同的：

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
        .05

SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
         .1
```

9.6.3 Simple 词典

Simple词典首先将输入标记转换为小写字母，然后检查停用词表。如果识别为停用词则返回空数组，即表示该标记会被丢弃。否则，输入标记的小写形式作为规范化后的lexeme返回。此外，Simple词典可通过设置参数Accept为false（默认值true），将非停用词报告为未识别，传递给后继词典继续处理。

注意事项

- 大多数词典的功能依赖于词典定义文件，词典定义文件名仅支持小写字母、数字、下划线组合。
- 临时模式pg_temp下不允许创建词典。
- 词典定义文件的字符集编码必须为UTF-8格式。实际应用时，如果与数据库的字符编码格式不一致，在读入词典定义文件时会进行编码转换。

- 通常情况下，每个session仅读取词典定义文件一次，当且仅当在第一次使用该词典时。需要修改词典文件时，可通过ALTER TEXT SEARCH DICTIONARY命令进行词典定义文件的更新和重新加载。

操作步骤

步骤1 创建Simple词典。

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

其中，停用词表文件全名为english.stop。关于创建simple词典的语法和更多参数，请参见**CREATE TEXT SEARCH DICTIONARY**。

步骤2 使用Simple词典。

```
SELECT ts_lexize('public.simple_dict','Yes');
 ts_lexize
-----------
 {yes}
(1 row)

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
(1 row)
```

步骤3 设置参数ACCEPT=false，使Simple词典返回NULL，而不是返回非停用词的小写形式。

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );
SELECT ts_lexize('public.simple_dict','Yes');
 ts_lexize
-----------

(1 row)

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
(1 row)
```

----结束

9.6.4 Synonym 词典

Synonym词典用于定义、识别token的同义词并转化，不支持词组（词组形式的同义词可用Thesaurus词典定义，详细请参见**Thesaurus词典**）。

示例

- Synonym词典可用于解决语言学相关问题，例如，为避免使单词"Paris"变成"pari"，可在Synonym词典文件中定义一行"Paris paris"，并将该词典放置在预定义的english_stem词典之前。

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}
(1 row)

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms,
    FILEPATH =  'obs://bucket01/obs.xxx.xxx.com accesskey=xxxxx secretkey=xxxxx region=cn-north-1'
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

SELECT * FROM ts_debug('english', 'paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

ALTER TEXT SEARCH DICTIONARY my_synonym ( CASESENSITIVE=true);

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
(1 row)

SELECT * FROM ts_debug('english', 'paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {pari}
(1 row)
```

其中，同义词词典文件全名为my_synonyms.syn，所在目录为 'obs://bucket01/obs.xxx.xxx.com accesskey=xxxxx secretkey=xxxxx region=*cn-north-1*'。关于创建词典的语法和更多参数，请参见**CREATE TEXT SEARCH DICTIONARY**。

- 星号（*）可用于词典文件中的同义词结尾，表示该同义词是一个前缀。在to_tsvector()中该星号将被忽略，但在to_tsquery()中会匹配该前缀并对应输出结果（参照**处理tsquery**一节）。

  假设词典文件synonym_sample.syn内容如下：

  ```
  postgres        pgsql
  postgresql      pgsql
  postgre pgsql
  gogle   googl
  indices index*
  ```

  创建并使用词典：

  ```
  CREATE TEXT SEARCH DICTIONARY syn (
      TEMPLATE = synonym,
      SYNONYMS = synonym_sample
  ```

```
);

SELECT ts_lexize('syn','indices');
 ts_lexize
-----------
 {index}
(1 row)

CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);

ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;

SELECT to_tsvector('tst','indices');
 to_tsvector
-------------
 'index':1
(1 row)

SELECT to_tsquery('tst','indices');
 to_tsquery
------------
 'index':*
(1 row)

SELECT 'indexes are very useful'::tsvector;
            tsvector
---------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');
 ?column?
----------
 t
(1 row)
```

9.6.5 Thesaurus 词典

Thesaurus词典，也叫做分类词典（缩写为TZ），是一组定义了词以及词组间关系的集合，包括广义词（BT）、狭义词（NT）、首选词、非首选词、相关词等。根据词典文件中的定义，TZ词典用一个指定的短语替换对应匹配的所有短语，并且可选择保留原始短语进行索引。TZ词典实际上是Synonym词典的一个扩展，增加了短语支持。

注意事项

- 由于TZ词典需要识别短语，所以在处理过程中必须保存当前状态并与解析器进行交互，以决定是否处理下一个token或是结束当前识别。此外，TZ词典配置时需谨慎，如果设置TZ词典仅处理asciiword类型的token，则类似one 7的分类词典定义将不会生效，因为uint类型的token不会传给TZ词典处理。
- 在索引期间要用到分类词典，因此分类词典参数中的任何变化都要求重新索引。对于其他大多数类型的词典来说，类似添加或删除停用词这种修改并不需要强制重新索引。

操作步骤

步骤1 创建一个名为thesaurus_astro的TZ词典。

以一个简单的天文学词典thesaurus_astro为例，其中定义了两组天文短语及其同义词如下：

```
supernovae stars : sn
crab nebulae : crab
```


执行如下语句创建TZ词典：

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = pg_catalog.english_stem,
    FILEPATH = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'
);
```

其中，词典定义文件全名为thesaurus_astro.ths，所在目录为 "obs://bucket_name/path accesskey=ak secretkey=sk region=rg"。子词典pg_catalog.english_stem是预定义的Snowball类型的英语词干词典，用于规范化输入词，子词典自身相关配置（例如停用词等）不在此处显示。关于创建词典的语法和更多参数，请参见**CREATE TEXT SEARCH DICTIONARY**。

步骤2 创建词典后，将其绑定到对应文本搜索配置中需要处理的token类型上：

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

步骤3 使用TZ词典。

- 测试TZ词典。

  ts_lexize函数对于测试TZ词典作用不大，因为该函数是按照单个token处理输入。可以使用plainto_tsquery、to_tsvector、to_tsquery函数测试TZ词典，这些函数能够将输入分解成多个token（to_tsquery函数需要将输入加上引号）。

  ```
  SELECT plainto_tsquery('english','supernova star');
   plainto_tsquery
  -----------------
   'sn'
  (1 row)

  SELECT to_tsvector('english','supernova star');
   to_tsvector
  -------------
   'sn':1
  (1 row)

  SELECT to_tsquery('english','''supernova star''');
   to_tsquery
  ------------
   'sn'
  (1 row)
  ```

  其中，supernova star匹配了词典thesaurus_astro定义中的supernovae stars，这是因为在thesaurus_astro词典定义中指定了Snowball类型的子词典english_stem，该词典移除了e和s。

- 如果同时需要索引原始短语，只要将其同时放置在词典定义文件中对应定义的右侧即可，如下：

  ```
  supernovae stars : sn supernovae stars

  ALTER TEXT SEARCH DICTIONARY thesaurus_astro (
      DictFile = thesaurus_astro,
      FILEPATH = 'file:///home/dicts/');

  SELECT plainto_tsquery('english','supernova star');
  ```

```
      plainto_tsquery
----------------------------
 'sn' & 'supernova' & 'star'
(1 row)
```

----结束

9.6.6 Ispell 词典

Ispell词典模板支持词法词典，它可以把一个词的各种语言学形式规范化成相同的词位。比如，一个Ispell英语词典可以匹配搜索词bank的词尾变化和词形变化，如banking、banked、banks、banks'和bank's等。

GaussDB(DWS)不提供任何预定义的Ispell类型词典或词典文件。dict文件和affix文件支持多种开源词典格式，包括Ispell、MySpell和Hunspell等。

操作步骤

步骤1 获取词典定义文件和词缀文件。

用户可以使用开源词典，直接获取的开源词典后缀名可能为.aff和.dic，此时需要将扩展名改为.affix和.dict。此外，对于某些词典文件，还需要使用下面的命令把字符转换成UTF-8编码，比如挪威语词典：

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

步骤2 创建Ispell词典。

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
CREATE TEXT SEARCH DICTIONARY norwegian_ispell (
    TEMPLATE = ispell,
    DictFile = nn_no,
    AffFile = nn_no,
    FilePath =  'obs://bucket01/obs.xxx.xxx.com accesskey=xxxxx secretkey=xxxxx region=cn-north-1'
);
```

其中，词典文件全名为nn_no.dict和nn_no.affix，所在目录为 'obs://bucket01/obs.xxx.xxx.com accesskey=xxxxx secretkey=xxxxx region=*cn-north-1*'。关于创建词典的语法和更多参数，请参见**CREATE TEXT SEARCH DICTIONARY**。

步骤3 使用Ispell词典进行复合词拆分。

```
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
      ts_lexize
---------------------
 {sjokolade,fabrikk}
(1 row)
```

MySpell不支持复合词，Hunspell对复合词有较好的支持。GaussDB(DWS)仅支持Hunspell中基本的复合词操作。通常情况下，Ispell词典能够识别的词是一个有限集合，其后应该配置一个更广义的词典，例如一个可以识别所有词的Snowball词典。

----结束

9.6.7 Snowball 词典

Snowball词典模板支持词干分析词典，基于Martin Porter的Snowball项目，内置有许多语言的词干分析算法。GaussDB(DWS)中预定义有多种语言的Snowball词典，可通过系统表PG_TS_DICT查看预定义的词干分析词典以及支持的语言词干分析算法。

无论是否可以简化，Snowball词典将标示所有输入为已识别，因此它应当被放置在词典列表的最后。把Snowball词典放在任何其他词典前面会导致后继词典失效，因为输入token不会通过Snowball词典进入到下一个词典。

关于Snowball词典的语法，请参见**CREATE TEXT SEARCH DICTIONARY**。

9.7 文本搜索配置示例

文本搜索配置（Text Search Configuration），指定了将文档转换成tsvector过程中所必需的组件：

- 解析器，用于把文本分解成标记token；
- 词典列表，用于将每个token转换成词位lexeme。

每次to_tsvector或to_tsquery函数调用时，都需要指定一个文本搜索配置来指定具体的处理过程。GUC参数default_text_search_config指定了默认的文本搜索配置，当文本搜索函数中没有显式指定文本搜索配置参数时，将会使用该默认值进行处理。

GaussDB(DWS)中预定义有一些可用的文本搜索配置，用户也可创建自定义的文本搜索配置。此外，为了便于管理文本搜索对象，还提供有多个gsql元命令，可以显示有关文本搜索对象的信息（详细请参见《工具指南》中“元命令参考”章节）。

操作步骤

步骤1 创建一个文本搜索配置ts_conf，复制预定义的文本搜索配置english。

```
CREATE TEXT SEARCH CONFIGURATION ts_conf ( COPY = pg_catalog.english );
CREATE TEXT SEARCH CONFIGURATION
```

步骤2 创建Synonym词典。

假设同义词词典定义文件pg_dict.syn内容如下：

```
postgres    pg
pgsql       pg
postgresql  pg
```

执行如下语句创建Synonym词典：

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict,
    FILEPATH =  'obs://bucket01/obs.xxx.xxx.com accesskey=xxxxx secretkey=xxxxx region=cn-north-1'
);
```

步骤3 创建一个Ispell词典english_ispell（词典定义文件来自开源词典）。

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english,
    FILEPATH =   'obs://bucket01/obs.xxx.xxx.com accesskey=xxxxx secretkey=xxxxx region=cn-north-1'
);
```

步骤4 设置文本搜索配置ts_conf，修改某些类型的token对应的词典列表。关于token类型的详细信息，请参见**文本搜索解析器**。

```
ALTER TEXT SEARCH CONFIGURATION ts_conf
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

步骤5 在文本搜索配置中，选择设置不索引或搜索某些token类型。

```
ALTER TEXT SEARCH CONFIGURATION ts_conf
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

步骤6 使用文本检索调测函数ts_debug()对所创建的词典配置ts_conf进行测试。

```
SELECT * FROM ts_debug('ts_conf', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

步骤7 可以设置当前session使用ts_conf作为默认的文本搜索配置。此设置仅在当前session有效。

```
\dF+ ts_conf
     Text search configuration "public.ts_conf"
Parser: "pg_catalog.default"
     Token      |            Dictionaries
-----------------+------------------------------------
 asciihword      | pg_dict,english_ispell,english_stem
 asciiword       | pg_dict,english_ispell,english_stem
 file            | simple
 host            | simple
 hword           | pg_dict,english_ispell,english_stem
 hword_asciipart | pg_dict,english_ispell,english_stem
 hword_numpart   | simple
 hword_part      | pg_dict,english_ispell,english_stem
 int             | simple
 numhword        | simple
 numword         | simple
 uint            | simple
 version         | simple
 word            | pg_dict,english_ispell,english_stem

SET default_text_search_config = 'public.ts_conf';
SET
SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.ts_conf
(1 row)
```

----结束

9.8 测试和调试文本搜索

9.8.1 分词器测试

函数ts_debug允许简单测试文本搜索分词器。

```
ts_debug([ config regconfig, ] document text,
        OUT alias text,
        OUT description text,
        OUT token text,
        OUT dictionaries regdictionary[],
        OUT dictionary regdictionary,
        OUT lexemes text[])
        returns setof record
```

ts_debug显示document的每个token信息，token是由解析器生成，由指定的词典进行处理。如果忽略对应参数，则使用config指定的分词器或者default_text_search_config指定的分词器。

ts_debug为文本解析器标识的每个token返回一行记录。记录中的列分别是：

- alias：text类型，token的别名。
- description：text类型，token的描述。
- token：text类型，token的文本内容。
- dictionaries：regdictionary数组类型，是分词器为token选定的词典。
- dictionary：regdictionary类型，用来识别token的词典。如果为空，则不做识别。
- lexemes：text数组类型，词典识别token时生成的词素。如果为空，则不生成词素。空数组（{}）意味着token将被识别成停用词。

一个简单的例子：

```
SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
(24 rows)
```

9.8.2 解析器测试

函数ts_parse可以直接测试文本搜索解析器。

```
ts_parse(parser_name text, document text,
        OUT tokid integer, OUT token text) returns setof record
```

ts_parse解析指定的document并返回一系列的记录，一条记录代表一个解析生成的token。每条记录包括标识token类型的tokid，及token文本。比如：

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
(6 rows)
ts_token_type(parser_name text, OUT tokid integer,
             OUT alias text, OUT description text) returns setof record
```

ts_token_type返回一个表，这个表描述了指定解析器可以识别的每种token类型。对于每个token类型，表中给出了整数类型的tokid--用于解析器标记对应的token类型；alias——命名分词器命令中的token类型；及简单描述。比如：

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
(23 rows)
```

9.8.3 词典测试

函数ts_lexize用于进行词典测试。

ts_lexize(dict regdictionary, token text) returns text[]如果输入的token可以被词典识别，那么ts_lexize返回词素的数组；如果token可以被词典识别但它是一个停用词，则返回空数组；如果是一个不可识别的词则返回NULL。

比如：

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
```

```
 ts_lexize
-----------
 {}
```

须知

ts_lexize函数支持单一token，不支持文本。

10 系统操作

GaussDB(DWS)通过SQL语句执行不同的系统操作，比如：设置变量，显示执行计划和垃圾收集等操作。

设置变量

设置会话或事务中需要使用的各种参数，请参考**SET**。

显示执行计划

显示GaussDB(DWS)为SQL语句规划的执行计划，请参考**EXPLAIN**。

事务日志检查点

预写式日志（WAL）缺省时在事务日志中每隔一段时间放置一个检查点。CHECKPOINT强制立即进行检查，而不是等到下一次调度时的检查点。请参考**CHECKPOINT**。

垃圾收集

进行垃圾收集以及可选择的对数据库进行分析。请参考**VACUUM**。

收集统计信息

收集与数据库中表内容相关的统计信息。请参考**ANALYZE | ANALYSE**。

设置当前事务的约束检查模式

设置当前事务里的约束检查的特性。请参考**SET CONSTRAINTS**。

11 事务管理

GaussDB(DWS)支持数据库事务ACID属性，提供了事务的读已提交隔离级别和可重复读隔离级别。

事务的概念

- **事务**指一个操作，由多个步骤组成，要么全部成功，要么全部失败。
- **数据库事务**（TRANSACTION）是数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成（通常由事务开始与事务结束之间执行的全部数据库操作组成），这些操作要么全部执行，要么全部不执行，是一个不可分割的执行单位。

事务的作用

数据库事务的目的主要是：

- 为数据库操作序列提供了一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法。
- 当多个应用程序在并发访问数据库时，可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。

事务的执行过程

当事务被提交给数据库管理系统（DBMS）后，DBMS需要确保该事务中的所有操作都成功完成，并且其结果被永久保存在数据库中。如果事务中有操作没有成功完成，则事务中的所有操作都需要回滚，回到事务执行前的状态；同时，该事务对数据库或者其他事务的执行无影响，所有的事务互不干扰和影响，好像在独立的环境中运行。

事务的属性

事务具有以下四个标准属性，通常根据首字母缩写为ACID。

- Atomicity（原子性）：事务中的全部操作在数据库中是不可分割的，整个事务中的所有操作要么全部完成，要么全部失败，对于一个事务来说，不能只执行其中的一部分操作。

  比如： A给B转账，A扣除500元 ，B增加500元。整个事务的操作要么全部成功，要么全部失败，不能出现A扣钱，但是B不增加的情况。如果原子性不能保证，就会很自然的出现一致性问题。

- Consistency（一致性）：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的数据必须完全符合所有的预设规则，这包含数据的精确度、串联性以及后续数据库可以自发性地完成预定的工作。

  比如：A给B转账，A扣除500元 ，B增加500元，扣除的钱-500与增加的钱+500，相加应该为0。如从A账户转账500元到B账户，不管操作成功与否，A和B的存款总额是不变的。

- Isolation（隔离性）：一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

  数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（serializable）。

- Durability（持久性）：一旦事务提交，则其所做的修改就会永久保存到数据库中。即使系统故障，已经提交的修改数据也不会丢失。

表 11-1 ACID 用途

ACID	属性	用途
Atomicity	原子性	并发控制，故障恢复。
Consistency	一致性	SQL的完整性约束（主键约束、外键约束）。
Isolation	隔离性	并发控制。
Durability	持久性	故障恢复。

常用的并发控制技术有基于锁的并发控制和基于时间戳的并发控制，GaussDB(DWS)数据库针对DDL语句采用两阶段锁技术，而针对DML语句则采用多版本控制技术（Multi-Version Concurrency Control，MVCC）。GaussDB(DWS)数据库的故障恢复采用WAL日志的方式来实现，目前主要支持Redo日志，通过Redo日志和MVCC可以保证事务读写的一致性。

隔离级别

Isolation（隔离性）可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离级别，决定多个事务并发操作同一个对象时的处理方式。

GaussDB(DWS)的事务隔离级别，由GUC参数transaction_isolation或**SET TRANSACTION**语法设置，支持以下隔离级别，默认为读已提交（read committed）。

- read committed：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。
- read uncommitted：读未提交隔离级别，GaussDB(DWS)不支持read uncommitted，如果设置了read uncommitted，实际上使用的是read committed。
- repeatable read：可重复读隔离级别，仅仅能看到事务开始之前提交的数据，不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。

- serializable：事务可序列化，GaussDB(DWS)不支持serializable，如果设置了serializable，实际上使用的是repeatable read。

事务控制语法

- 启动事务

  GaussDB(DWS)通过START TRANSACTION和BEGIN语法启动事务，请参考**START TRANSACTION**和**BEGIN**。

- 设置事务

  GaussDB(DWS)通过SET TRANSACTION或者SET LOCAL TRANSACTION语法设置事务，请参考**SET TRANSACTION**。

- 提交事务

  GaussDB(DWS)通过COMMIT或者END可完成提交事务的功能，即提交事务的所有操作，请参考**COMMIT | END**。

- 回滚事务

  回滚是在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销。请参考**ROLLBACK**。

  说明

  数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务，如果其中有一个语句失败，那么整个请求都将会被回滚。

- 其他事务操作
  - SAVEPOINT用于在当前事务里建立一个新的保存点。即在一个事务中标记一个位置并且允许做部分回滚。用户可以回滚在一个保存点之后执行的命令但保留该保存点之前执行的命令。请参考**SAVEPOINT**。
  - ROLLBACK TO SAVEPOINT回滚事务到一个保存点。隐含地删除所有在该保存点之后建立的保存点。请参考**ROLLBACK TO SAVEPOINT**。
  - RELEASE SAVEPOINT删除一个事务内的保存点。请参考**RELEASE SAVEPOINT**。

事务场景示例

某顾客在商店使用电子支付购买100元的物品，当中至少包括两个操作：1. 该顾客的账户减少100元。2. 商店账户（商户）增加100元。支持事务的数据库管理系统就是要确保以上两个操作（整个“事务”）都能完成，或一起取消。

1. 创建样例数据：

   创建一个简单的用户金额表并向表中插入数据（假设商户和顾客的账户上各有500元）。

   ```
   CREATE TABLE customer_info (
       NAME VARCHAR(32) PRIMARY KEY,
       MONEY INTEGER
   );
   INSERT INTO customer_info (name, money) VALUES ('buyer', 500), ('shop', 500);
   ```

   查看表数据显示商户和顾客各有500元。

   ```
   SELECT * FROM customer_info;
    name  | money
   -------+-------
    buyer |   500
    shop  |   500
   (2 rows)
   ```

2. 普通操作（正常模式）。

   模拟正常购买过程，顾客先扣款100元，商户再增加款额100元。

```
UPDATE customer_info SET money = money-100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'buyer');
UPDATE customer_info SET money = money+100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'shop');

SELECT * FROM customer_info;
 name  | money
-------+-------
 buyer |   400
 shop  |   600
(2 rows)
```

3. 恢复初始值。

```
UPDATE customer_info SET money=500;
select * from customer_info;
 name  | money
-------+-------
 shop  |   500
 buyer |   500
(2 rows)
```

4. 普通操作（异常模式）。

   模拟购买过程出现状况，顾客发生扣款100元，商户没有增加款额。

   a. 顾客先扣款100元。

```
UPDATE customer_info SET money = money-100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'buyer');
```

   b. 商户发现支付有问题，终止了后续交易。商户增加款操作直接报错，终止执行下面的语句。（仅商户觉得支付有问题）

```
UPDATE customer_info SET money = money+100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'shop');
```

   c. 查询结果发现：消费者已经扣款，但商户没增加款额，这里顾客的金额了100元。

```
SELECT * FROM customer_info;
 name  | money
-------+-------
 buyer |   400
 shop  |   500
(2 rows)
```

因此，如果没有事务，一旦SQL语句中间出现异常，整个账户系统的收支就不平衡了。

使用数据库事务，模拟出现异常操作时，进行事务回滚。

1. 恢复初始值：

```
UPDATE customer_info SET money=500;
```

2. 开启事务后，顾客先扣款100元。

```
BEGIN TRANSACTION;
UPDATE customer_info SET money = money-100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'buyer');
```

3. 商户增加款额操作直接报错，终止执行下面的语句。

```
UPDATE customer_info SET money = money+100 WHERE name IN (SELECT name FROM
customer_info WHERE name = 'shop');
```

4. 回滚事务，在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销。

```
ERROR:  syntax error at or near "shop"
LINE 1: ...e IN (SELECT name FROM customer_info WHERE name = ''shop'');
END TRANSACTION;
ROLLBACK
```


5. 查询显示顾客和商户的账户金额仍旧完整一致。即数据库在事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，数据库的完整性没有被破坏。

```
SELECT * FROM customer_info;
 name  | money
-------+-------
 buyer |   500
 shop  |   500
(2 rows)
```

两阶段事务

GaussDB(DWS)属于分布式share-nothing架构，表的数据分布在不同的节点上。客户端的一条或多条语句可能会同时修改多个节点上的数据，这种情况下，会产生分布式事务。GaussDB(DWS)采用两阶段提交事务来保证分布式事务中数据的一致性和事务的原子性。顾名思义，两阶段提交就是将事务提交划分为两个阶段，通常针对的是包含写操作的事务。当写操作将数据写入不同的节点时，需要满足事务的原子性要求，要么全部提交，要么全部回滚。

不支持两阶段的场景如下：

- 不支持显示的两阶段提交语法PREPARE TRANSACTION。

  ```
  BEGIN;
  PREPARE TRANSACTION 'p1';
  ERROR：Explicit prepare transaction is not supported.
  ```

- 不支持在两阶段事务中修改系统表的文件映射关系。

  ```
  REINDEX TABLE pg_class;
  ERROR：cannot PREPARE a transaction that modified relation mapping.
  ```

- 不支持在跨节点的事务中提交导出事务快照。

  ```
  BEGIN;
  CREATE TABLE t1(a int);
  SELECT pg_export_snapshot();
  END;
  ERROR：cannot PREPARE a transaction that has exported snapshots.
  ```


12 DDL 语法

12.1 DDL 语法一览表

DDL（Data Definition Language数据定义语言），用于定义或修改数据库中的对象。如：表、索引、视图等。

说明

GaussDB(DWS)不支持CN不完整时进行DDL操作。例如：集群中有1个CN故障时执行新建数据库、表等操作都会失败。

定义数据库

数据库是组织、存储和管理数据的仓库，而数据库定义主要包括：创建数据库、修改数据库属性，以及删除数据库。所涉及的SQL语句，请参考下表。

表 12-1 数据库定义相关 SQL

功能	相关SQL
创建数据库	CREATE DATABASE
修改数据库属性	ALTER DATABASE
删除数据库	DROP DATABASE

定义模式

模式是一组数据库对象的集合，主要用于控制对数据库对象的访问。所涉及的SQL语句，请参考下表。

表 12-2 模式定义相关 SQL

功能	相关SQL
创建模式	CREATE SCHEMA

功能	相关SQL
修改模式属性	ALTER SCHEMA
删除模式	DROP SCHEMA

定义表

表是数据库中的一种特殊数据结构，用于存储数据对象以及对象之间的关系。所涉及的SQL语句，请参考下表。

表 12-3 表定义相关 SQL

功能	相关SQL
创建表	CREATE TABLE
修改表属性	ALTER TABLE
修改表名	RENAME TABLE
删除表	DROP TABLE
删除表的所有数据	TRUNCATE

定义分区表

分区表是一种逻辑表，数据是由普通表存储的，主要用于提升查询性能。所涉及的SQL语句，请参考下表。

表 12-4 分区表定义相关 SQL

功能	相关SQL
创建分区表	CREATE TABLE PARTITION
修改分区	ALTER TABLE PARTITION
修改分区表属性	ALTER TABLE PARTITION
删除分区	ALTER TABLE PARTITION
删除分区表	DROP TABLE

定义外表

外表是指一个逻辑上的表对象，它对应的实际数据存储位置并不在数据库内部，而是存储于外部存储服务中。

表 12-5 外表定义相关 SQL

功能	相关SQL
创建GDS外表	CREATE FOREIGN TABLE (GDS导入导出)
创建HDFS或OBS外表（需手动创建Server）	CREATE FOREIGN TABLE (SQL on OBS or Hadoop)
创建OBS外表（默认Server）	CREATE FOREIGN TABLE (OBS导入导出)
创建协同分析外表	CREATE FOREIGN TABLE (SQL on other GaussDB(DWS))
修改GDS外表	ALTER FOREIGN TABLE (GDS导入导出)
HDFS外表和OBS外表	ALTER FOREIGN TABLE (For HDFS or OBS)
修改协同分析外表	ALTER FOREIGN TABLE (SQL on other GaussDB(DWS))
删除外表	DROP FOREIGN TABLE

定义索引

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。所涉及的SQL语句，请参考下表。

表 12-6 索引定义相关 SQL

功能	相关SQL
创建索引	CREATE INDEX
修改索引属性	ALTER INDEX
删除索引	DROP INDEX
重建索引	REINDEX

定义角色

角色是用来管理权限的，从数据库安全的角度考虑，可以把所有的管理和操作权限划分到不同的角色上。所涉及的SQL语句，请参考下表。

表 12-7 角色定义相关 SQL

功能	相关SQL
创建角色	CREATE ROLE
修改角色属性	ALTER ROLE
删除角色	DROP ROLE

定义用户

用户是用来登录数据库的，通过对用户赋予不同的权限，可以方便地管理用户对数据库的访问及操作。所涉及的SQL语句，请参考下表。

表 12-8 用户定义相关 SQL

功能	相关SQL
创建用户	CREATE USER
修改用户属性	ALTER USER
删除用户	DROP USER

定义脱敏策略

数据脱敏策略是指对某些敏感信息通过脱敏规则进行数据的变形，实现敏感隐私数据的可靠保护。用户可以在指定表对象创建脱敏策略，并限定策略生效范围，也可以新增、修改、删除脱敏列信息。所涉及的SQL语句，请参考下表。

表 12-9 脱敏策略定义相关 SQL

功能	相关SQL
对表创建数据脱敏策略	CREATE REDACTION POLICY
修改应用在指定表的脱敏策略	ALTER REDACTION POLICY
删除应用在指定表的脱敏策略	DROP REDACTION POLICY

定义行级访问控制

行级访问控制策略控制数据库表中行级数据可见性。不同用户执行相同的SQL查询操作，读取到的结果不同。所涉及的SQL语句，请参考下表。

表 12-10 行级访问控制定义相关 SQL

功能	相关SQL
创建行访问控制策略	CREATE ROW LEVEL SECURITY POLICY
修改已存在的行访问控制策略	ALTER ROW LEVEL SECURITY POLICY
删除表上某个行访问控制策略	DROP ROW LEVEL SECURITY POLICY

定义存储过程

存储过程是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。所涉及的SQL语句，请参考下表。

表 12-11 存储过程定义相关 SQL

功能	相关SQL
创建存储过程	CREATE PROCEDURE
删除存储过程	DROP PROCEDURE

定义函数

在GaussDB(DWS)中，它和存储过程类似，也是一组SQL语句集，使用上没有差别。所涉及的SQL语句，请参考下表。

表 12-12 函数定义相关 SQL

功能	相关SQL
创建函数	CREATE FUNCTION
修改函数属性	ALTER FUNCTION
删除函数	DROP FUNCTION

定义视图

视图是从一个或几个基本表中导出的虚表，可用于控制用户对数据访问，请参考下表。

表 12-13 视图定义相关 SQL

功能	相关SQL
创建视图	CREATE VIEW
删除视图	DROP VIEW

定义序列

序列属于数据库对象，用户可以从序列中生成唯一整数。

表 12-14 序列定义相关 SQL

功能	相关SQL
创建序列	CREATE SEQUENCE
修改序列	ALTER SEQUENCE
删除序列	DROP SEQUENCE

定义触发器

触发器是一种特殊的存储过程，它与数据库表的事件相关联，可以在这些事件发生时自动执行，由对表的某些操作（如插入、删除、更新）所触发。

表 12-15 触发器定义相关 SQL

功能	相关SQL
创建触发器	CREATE TRIGGER
修改触发器	ALTER TRIGGER
删除触发器	DROP TRIGGER

定义游标

为了处理SQL语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化，请参考下表。

表 12-16 游标定义相关 SQL

功能	相关SQL
创建游标	CURSOR
移动游标	MOVE

功能	相关SQL
从游标中提取数据	FETCH
关闭游标	CLOSE

操作会话

用户与数据库之间建立的连接称为会话，请参考下表。

表 12-17 会话相关 SQL

功能	相关SQL
修改会话	ALTER SESSION
结束会话	ALTER SYSTEM KILL SESSION

定义资源池

资源池是负载管理模块使用的系统表，主要用于指定资源管理相关的属性，如控制组。所涉及的SQL语句，请参考下表。

表 12-18 资源池定义相关 SQL

功能	相关SQL
创建资源池	CREATE RESOURCE POOL
修改资源池属性	ALTER RESOURCE POOL
删除资源池	DROP RESOURCE POOL

定义同义词

同义词是兼容Oracle的一种特殊的数据库对象，用于存储与另一个数据库对象名间的映射关系，目前仅支持使用同义词关联以下数据库对象：表、视图、函数和存储过程。所涉及的SQL语句，请参考下表。

表 12-19 同义词定义相关 SQL

功能	相关SQL
创建同义词	CREATE SYNONYM
修改同义词	ALTER SYNONYM
删除同义词	DROP SYNONYM

定义文本搜索配置

文本搜索配置指定了文本搜索解析器，该文本搜索解析器可以将字符串划分为标记，外加一些词典（可被用来决定哪些标记是搜索感兴趣的）。所涉及的SQL语句，请参考下表。

表 12-20 文本搜索配置相关 SQL

功能	相关SQL
创建文本搜索配置	CREATE TEXT SEARCH CONFIGURATION
修改文本搜索配置	ALTER TEXT SEARCH CONFIGURATION
删除文本搜索配置	DROP TEXT SEARCH CONFIGURATION

定义全文检索词典

词典是在全文检索时识别特定词并进行处理。词典的创建依赖于预定义模板（在系统表PG_TS_TEMPLATE中定义），支持创建五种类型的词典，分别是Simple、Ispell、Synonym、Thesaurus、以及Snowball，每种类型的词典可以完成不同的任务。所涉及的SQL语句，请参考下表。

表 12-21 全文检索词典相关 SQL

功能	相关SQL
创建全文检索词典	CREATE TEXT SEARCH DICTIONARY
修改全文检索词典	ALTER TEXT SEARCH DICTIONARY
删除全文检索词典	DROP TEXT SEARCH DICTIONARY

12.2 ALTER DATABASE

功能描述

修改数据库的属性，包括它的名称、所有者、连接数限制、对象隔离属性等。

注意事项

- 只有拥有数据库所有者权限的用户才能执行ALTER DATABASE命令，系统管理员默认拥有此权限。如果是非系统管理员，针对所要修改属性的不同，对其还有以下权限约束：
  - 修改数据库名称，必须拥有CREATEDB权限。
  - 修改数据库所有者，当前用户必须是该数据库的所有者且拥有CREATEDB权限，并确保该用户是新所有者角色的成员。

  - 修改数据库默认表空间，该用户必须是该数据库的所有者或系统管理员且拥有新表空间的CREATE权限。该语法从物理上将一个数据库原来缺省表空间上的表和索引移至新的表空间。注意不在缺省表空间的表和索引不受此影响。
  - 修改某个按数据库设置的相关参数，只有数据库所有者或者系统管理员可以改变这些设置。
  - 修改某个数据库对象隔离属性，只有数据库所有者或者系统管理员可以执行此操作。
- 不能重命名当前使用的数据库，如果需要重新命名，须连接至其他数据库上。
- 不支持修改现有数据库的兼容模式，只能在创建数据库时指定兼容模式，详情请参见**CREATE DATABASE**。

语法格式

- 修改数据库的最大连接数。
```
ALTER DATABASE database_name
    [ [ WITH ] CONNECTION LIMIT connlimit ];
```
- 修改数据库名称。
```
ALTER DATABASE database_name
    RENAME TO new_name;
```

  说明

  若该数据库中有OBS冷热表，则不支持修改数据库名。
- 修改数据库所属者。
```
ALTER DATABASE database_name
    OWNER TO new_owner;
```
- 修改数据库默认表空间。
```
ALTER DATABASE database_name
    SET TABLESPACE new_tablespace;
```

  说明

  修改数据库的表空间时不能修改为OBS表空间。
- 修改数据库指定会话参数值。
```
ALTER DATABASE database_name
    SET configuration_parameter { { TO | = } { value | DEFAULT } | FROM CURRENT };
```
- 数据库配置参数重置。
```
ALTER DATABASE database_name RESET
    { configuration_parameter | ALL };
```
- 修改数据库对象隔离属性。
```
ALTER DATABASE database_name [ WITH ] { ENABLE | DISABLE } PRIVATE OBJECT;
```

  说明

  - 修改数据库的对象隔离属性时须连接至该数据库，否则无法更改。
  - 新创建的数据库，对象隔离属性默认是关闭的。当开启数据库对象隔离属性后，普通用户只能查看有权访问的对象（表、函数、视图、字段等）。对象隔离特性对管理员用户不生效，当开启对象隔离特性后，管理员也可以查看到全量的数据库对象。

参数说明

- **database_name**

  需要修改属性的数据库名称。

  取值范围：字符串，要符合标识符的命名规范。

- **connlimit**

  数据库可以接收的最大并发连接数（管理员用户连接除外）。

  取值范围：整数，建议填写1~50的整数。-1（缺省）表示没有限制。

- **new_name**

  数据库的新名称。

  取值范围：字符串，要符合标识符的命名规范。

- **new_owner**

  数据库的新所有者。

  取值范围：字符串，有效的用户名。

- **configuration_parameter**

  value

  把指定的数据库会话参数值设置为给定的值。如果value是DEFAULT或者RESET，则在新的会话中使用系统的缺省设置。OFF关闭设置。

  取值范围：字符串，

  - DEFAULT
  - OFF
  - RESET

- **FROM CURRENT**

  根据当前会话连接的数据库设置该参数的值。

- **RESET configuration_parameter**

  重置指定的数据库会话参数值。

- **RESET ALL**

  重置全部的数据库会话参数值。

说明

- 修改数据库默认表空间，会将旧表空间中的所有表和索引转移到新表空间中，该操作不会影响其他非默认表空间中的表和索引。
- 修改的数据库会话参数值，将在下一次会话中生效。

示例

创建示例数据库testdb和用户tom：

```
CREATE DATABASE testdb ENCODING 'UTF8' template = template0;
DROP USER IF EXISTS tom;
CREATE USER tom PASSWORD '{Password}';
```

设置testdb数据库的连接数为10：

```
ALTER DATABASE testdb CONNECTION LIMIT= 10;
```

将testdb名称改为testdb_1：

```
ALTER DATABASE testdb RENAME TO testdb_1;
```

将数据库testdb_1的所属者改为tom：

```
ALTER DATABASE testdb_1 OWNER TO tom;
```

设置music1的表空间为PG_DEFAULT：


```
ALTER DATABASE testdb_1 SET TABLESPACE PG_DEFAULT;
```

关闭在数据库testdb_1上缺省的索引扫描：

```
ALTER DATABASE testdb_1 SET enable_indexscan TO off;
```

重置enable_indexscan参数：

```
ALTER DATABASE testdb_1 RESET enable_indexscan;
```

相关链接

CREATE DATABASE，**DROP DATABASE**

12.3 ALTER FOREIGN TABLE (GDS 导入导出)

功能描述

对外表进行修改。

注意事项

无。

语法格式

- 设置外表属性

```
ALTER FOREIGN TABLE [ IF EXISTS ]  table_name
    OPTIONS ( {[ ADD | SET | DROP ] option ['value']}[, ... ]);
```

- 设置新的所有者

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    OWNER TO new_owner;
```

参数说明

- **table_name**

  需要修改的外表名称。

  取值范围：已存在的外表名。

- **option**

  需要修改的option名称。

  取值范围：请参见CREATE FOREIGN TABLE的**参数说明**。

- **value**

  option的新值。

示例

创建外表customer_ft，用来以TEXT格式导入GDS服务器10.10.123.234上的数据：

```
DROP FOREIGN TABLE IF EXISTS customer_ft;
CREATE FOREIGN TABLE customer_ft
(
    c_customer_sk             integer               ,
    c_customer_id             char(16)              ,
    c_current_cdemo_sk        integer               ,
```

```
    c_current_hdemo_sk        integer             ,
    c_current_addr_sk         integer             ,
    c_first_shipto_date_sk    integer             ,
    c_first_sales_date_sk     integer             ,
    c_salutation              char(10)            ,
    c_first_name              char(20)            ,
    c_last_name               char(30)            ,
    c_preferred_cust_flag     char(1)             ,
    c_birth_day               integer             ,
    c_birth_month             integer             ,
    c_birth_year              integer                   ,
    c_birth_country           varchar(20)               ,
    c_login                   char(13)                  ,
    c_email_address           char(50)                  ,
    c_last_review_date        char(10)
)
    SERVER gsmpp_server
    OPTIONS
(
    location 'gsfs://10.10.123.234:5000/customer1*.dat',
    FORMAT 'TEXT' ,
    DELIMITER '|',
    encoding 'utf8',
    mode 'Normal')READ ONLY;
```

修改外表customer_ft属性，删除mode选项：

```
ALTER FOREIGN TABLE customer_ft options(drop mode);
```

相关链接

CREATE FOREIGN TABLE (GDS导入导出)，**DROP FOREIGN TABLE**

12.4 ALTER FOREIGN TABLE (For HDFS or OBS)

功能描述

对HDFS外表和OBS外表进行修改。

注意事项

无。

语法格式

- 设置外表属性：

```
ALTER FOREIGN TABLE [ IF EXISTS ]  table_name
    OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ]);
```

- 设置外表的所有者：

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    OWNER TO new_owner;
```

- 更新外表列：

```
ALTER FOREIGN TABLE [ IF EXISTS ] table_name
    MODIFY ( { column_name data_type | column_name [ CONSTRAINT constraint_name ] NOT NULL
[ ENABLE ] | column_name [ CONSTRAINT constraint_name ] NULL } [, ...] );
```

- 修改外表的列：

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    action [, ... ];
```

其中action语法为：


```
ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type
  | ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
  | ALTER [ COLUMN ] column_name SET STATISTICS [PERCENT] integer
  | ALTER [ COLUMN ] column_name OPTIONS ( {[ ADD | SET | DROP ] option ['value'] } [, ... ])
  | MODIFY column_name data_type
  | MODIFY column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ]
  | MODIFY column_name [ CONSTRAINT constraint_name ] NULL
```

参考**ALTER TABLE**。

- 增加外表信息约束（Informational Constraint）的语法为：

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
   ADD [ CONSTRAINT constraint_name ]
   { PRIMARY KEY | UNIQUE } ( column_name )
   [ NOT ENFORCED [ ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION ] |
ENFORCED ];
```

对于增加外表信息约束（Informational Constraint）相关参数请参考CREATE FOREIGN TABLE （For HDFS）的**参数说明**。

- 删除外表的信息约束（Informational Constraint）语法为：

```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
   DROP CONSTRAINT constraint_name ;
```

参数说明

- **IF EXISTS**

  如果不存在相同名称的表，不会抛出错误，而会返回一个通知，告知表不存在。

- **tablename**

  需要修改的外表名称。

  取值范围：已存在的外表名。

- **new_owner**

  外表的新所有者。

  取值范围：字符串，有效的用户名。

- **data_type**

  现存字段的新类型。

  取值范围：字符串，需符合标识符的命名规范。

- **constraint_name**

  要添加/删除的约束的名称。

- **column_name**

  现存字段的名称。

  取值范围：字符串，需符合标识符的命名规范。

修改外表语法中其他参数如IF EXISTS，请参见ALTER TABLE的**参数说明**。

示例

1. 创建HDFS_Server，对应的foreign data wrapper为HDFS_FDW或者DFS_FDW。

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS (address
'10.10.0.100:25000,10.10.0.101:25000',hdfscfgpath '/opt/hadoop_client/HDFS/hadoop/etc/
hadoop',type'HDFS');
```

2. 创建不包含分区列的HDFS外表：

```
DROP FOREIGN TABLE IF EXISTS ft_region;
CREATE FOREIGN TABLE ft_region
(
   R_REGIONKEY INT4,
```

```
    R_NAME TEXT,
    R_COMMENT TEXT
)
SERVER
    hdfs_server
OPTIONS
(
    FORMAT 'orc',
    encoding 'utf8',
    FOLDERNAME '/user/hive/warehouse/gauss.db/region_orc11_64stripe/'
)
DISTRIBUTE BY
     roundrobin;
```

3. 修改外表ft_region的r_name字段类型为text：
```
ALTER FOREIGN TABLE ft_region ALTER r_name TYPE TEXT;
```
4. 标记外表ft_region的r_name列为not null：
```
ALTER FOREIGN TABLE ft_region ALTER r_name SET NOT NULL;
```

相关链接

CREATE FOREIGN TABLE (SQL on OBS or Hadoop)，DROP FOREIGN TABLE

12.5 ALTER FOREIGN TABLE (SQL on other GaussDB(DWS))

功能描述

对协同分析的外表进行修改。

注意事项

无。

语法格式

- 设置外表属性：
```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    OPTIONS ( {[ SET ] option ['value']} [, ... ]);
```
- 设置外表的所有者：
```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    OWNER TO new_owner;
```
- 更新外表列类型：
```
ALTER FOREIGN TABLE [ IF EXISTS ] table_name
    MODIFY ( { column_name data_type [, ...] } );
```
- 修改外表的列：
```
ALTER FOREIGN TABLE [ IF EXISTS ] tablename
    action [, ... ];
```
其中action语法为：
```
ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type
    | MODIFY column_name data_type
```
参考ALTER TABLE。

参数说明

- **IF EXISTS**

如果不存在相同名称的表，不会抛出错误，而会返回一个通知，告知表不存在。

- **tablename**
  需要修改的外表名称。
  取值范围：已存在的外表名。
- **new_owner**
  外表的新所有者。
  取值范围：字符串，有效的用户名。
- **data_type**
  现存字段的新类型。
  取值范围：字符串，需符合标识符的命名规范。
- **column_name**
  现存字段的名称。
  取值范围：字符串，需符合标识符的命名规范。

修改外表语法中其它参数，请参见ALTER TABLE的**参数说明**。

示例

1. 创建名称为server_remote的外部服务器，对应的foreign data wrapper为GC_FDW。
```
CREATE SERVER server_remote FOREIGN DATA WRAPPER GC_FDW OPTIONS (address
'10.10.0.100:25000,10.10.0.101:25000',dbname 'test', username 'test', password '{Password}');
```
2. 创建名称为region的外表：
```
DROP FOREIGN TABLE IF EXISTS region;
CREATE FOREIGN TABLE region
(
    R_REGIONKEY INT4,
    R_NAME TEXT,
    R_COMMENT TEXT
)
SERVER
    server_remote
OPTIONS
(
    schema_name 'test',
    table_name 'region',
    encoding 'gbk'
);
```
3. 修改外表region选项：
```
ALTER FOREIGN TABLE region OPTIONS (SET schema_name 'test');
```
4. 修改外表region的r_name字段类型为text：
```
ALTER FOREIGN TABLE region ALTER r_name TYPE TEXT;
```

相关链接

DROP FOREIGN TABLE，**CREATE FOREIGN TABLE (SQL on other GaussDB(DWS))**

12.6 ALTER FUNCTION

功能描述

修改自定义函数的属性。

注意事项

只有该函数的所有者，才有权限执行该命令，系统管理员默认拥有该权限。要修改函数的所有者的用户必须是新拥有角色的直接或间接成员。如果函数中涉及对临时表相关的操作，则无法使用ALTER FUNCTION。

语法格式

- 修改自定义函数的附加参数：

```
ALTER FUNCTION function_name ( [ { [ argmode ] [ argname ] argtype} [, ...] ] )
    action [ ... ] [ RESTRICT ];
```

  其中附加参数action子句语法为：

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
 | {IMMUTABLE | STABLE | VOLATILE}
 | {SHIPPABLE | NOT SHIPPABLE}
 | {NOT FENCED | FENCED}
 | [ NOT ] LEAKPROOF
 | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
 | AUTHID { DEFINER | CURRENT_USER }
 | COST execution_cost
 | ROWS result_rows
 | SET configuration_parameter { { TO | = } { value | DEFAULT }| FROM CURRENT}
 | RESET {configuration_parameter | ALL}
```

- 修改自定义函数的名字：

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype} [, ...] ] )
    RENAME TO new_name;
```

- 修改自定义函数的所属者：

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype} [, ...] ] )
    OWNER TO new_owner;
```

- 修改自定义函数的模式：

```
ALTER FUNCTION funname ( [ { [ argmode ] [ argname ] argtype} [, ...] ] )
    SET SCHEMA new_schema;
```

参数说明

- **function_name**

  要修改的函数名称。

  取值范围：已存在的函数名。

- **argmode**

  标识该参数是输入、输出参数。

  取值范围：IN/OUT/IN OUT

- **argname**

  参数名称。

  取值范围：字符串，符合标识符命名规范。

- **argtype**

参数类型。

取值范围：有效的类型，请参考**数据类型**。

- **CALLED ON NULL INPUT**

  表明该函数的某些参数是NULL的时候可以按照正常的方式调用。缺省时与指定此参数的作用相同。

- **RETURNS NULL ON NULL INPUT**

  STRICT

  STRICT用于指定如果函数的某个参数是NULL，此函数总是返回NULL。如果声明了该参数，则如果存在NULL参数时就不会执行此函数，而是自动假设一个NULL结果。

  RETURNS NULL ON NULL INPUT和STRICT的功能相同。

- **IMMUTABLE**

  表示该函数在给出同样的参数值时总是返回同样的结果。

- **STABLE**

  表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同SQL语句之间变化。

- **VOLATILE**

  表示该函数值可以在一次表扫描内改变，不会做任何优化。

- **SHIPPABLE**

  NOT SHIPPABLE

  表示该函数是否可以下推到DN上执行。

  - 对于IMMUTABLE类型的函数，函数始终可以下推到DN上执行。
  - 对于STABLE/VOLATILE类型的函数，仅当函数的属性是SHIPPABLE的时候，函数可以下推到DN执行。

- **LEAKPROOF**

  表示该函数没有副作用，指出参数只包括返回值。LEAKPROOF只能由系统管理员设置。

- **（可选）EXTERNAL**

  目的是和SQL兼容，这个特性适合于所有函数，而不仅是外部函数。

- **SECURITY INVOKER**

  AUTHID CURREN_USER

  表明该函数将以调用它的用户的权限执行。缺省时与指定此参数的作用相同。

  SECURITY INVOKER和AUTHID CURREN_USER的功能相同。

- **SECURITY DEFINER**

  AUTHID DEFINER

  声明该函数将以创建它的用户的权限执行。

  AUTHID DEFINER和SECURITY DEFINER的功能相同。

- **COST execution_cost**

  用来估计函数的执行成本。

  execution_cost以cpu_operator_cost为单位。

  取值范围：正数

- **ROWS result_rows**

  估计函数返回的行数。用于函数返回的是一个集合。

  取值范围：正数，默认值是1000行。

- **configuration_parameter**
  - **value**

    把指定的数据库会话参数值设置为给定的值。如果value是DEFAULT或者RESET，则在新的会话中使用系统的缺省设置。OFF关闭设置。

    取值范围：字符串

    - DEFAULT
    - OFF
    - RESET

    指定默认值。
  - **from current**

    取当前会话中的值设置为configuration_parameter的值。
- **new_name**

  函数的新名称。要修改函数的所属模式，必须拥有新模式的CREATE权限。

  取值范围：字符串，符合标识符命名规范。

- **new_owner**

  函数的新所有者。要修改函数的所有者，新所有者必须拥有该函数所属模式的CREATE权限。

  取值范围：已存在的用户角色。

- **new_schema**

  函数的新模式。

  取值范围：已存在的模式。

示例

创建计算两个整数的和的函数，并返回结果。若果输入为null，则返回null：

```
DROP FUNCTION IF EXISTS func_add_sql2;
CREATE FUNCTION func_add_sql2(num1 integer, num2 integer) RETURN integer
AS
BEGIN
RETURN num1 + num2;
END;
/
```

修改函数add的执行规则为IMMUTABLE，即参数不变时返回相同结果：

```
ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) IMMUTABLE;
```

将函数func_add_sql2的名称修改为add_two_number：

```
ALTER FUNCTION func_add_sql2(INTEGER, INTEGER) RENAME TO add_two_number;
```

将函数add_two_number的属者改为dbadmin：

```
ALTER FUNCTION add_two_number(INTEGER, INTEGER) OWNER TO dbadmin;
```


相关链接

CREATE FUNCTION，DROP FUNCTION

12.7 ALTER GROUP

功能描述

修改一个用户组的属性。

注意事项

ALTER GROUP是ALTER ROLE的别名，非SQL标准语法，不推荐使用，建议用户直接使用ALTER ROLE替代。

语法格式

- 向用户组中添加用户。
```
ALTER GROUP group_name
    ADD USER user_name [, ... ];
```
- 从用户组中删除用户。
```
ALTER GROUP group_name
    DROP USER user_name [, ... ];
```
- 修改用户组的名称。
```
ALTER GROUP group_name
    RENAME TO new_name;
```

参数说明

请参考ALTER ROLE的参数说明。

相关链接

CREATE GROUP，DROP GROUP，ALTER ROLE

12.8 ALTER INDEX

功能描述

ALTER INDEX用于修改现有索引的定义。

注意事项

- 只有索引的所有者有权限执行此命令，系统管理员默认拥有此权限。

语法格式

- 重命名表索引的名字。
```
ALTER INDEX [ IF EXISTS ] index_name
    RENAME TO new_name;
```
- 修改表索引的存储参数。
```
ALTER INDEX [ IF EXISTS ] index_name
    SET ( {storage_parameter = value} [, ... ] );
```

- 重置表索引的存储参数。
```
ALTER INDEX [ IF EXISTS ] index_name
    RESET ( storage_parameter [, ... ] ) ;
```
- 设置表索引或索引分区不可用。
```
ALTER INDEX [ IF EXISTS ] index_name
    [ MODIFY PARTITION index_partition_name ] UNUSABLE;
```

  说明

  列存表不支持该语法。

- 重建表索引或索引分区。
```
ALTER INDEX index_name
    REBUILD [ PARTITION index_partition_name ];
```
- 重命名索引分区。
```
ALTER INDEX [ IF EXISTS ] index_name
    RENAME PARTITION index_partition_name TO new_index_partition_name;
```

  说明

  PG_OBJECT系统表记录索引最后修改时间时不支持对该语法的记录。

- 添加，修改索引的注释。
```
ALTER INDEX [ IF EXISTS ] index_name
    COMMENT 'text';
```
- 删除索引的注释。
```
ALTER INDEX [ IF EXISTS ] index_name
    COMMENT '';
ALTER INDEX [ IF EXISTS ] index_name
    COMMENT NULL;
```

参数说明

- **IF EXISTS**

  如果指定的索引不存在，则发出一个notice而不是error。

- **RENAME TO**

  只改变索引的名字。对存储的数据没有影响。

- **SET ({ STORAGE_PARAMETER = value } [, ...])**

  改变索引的一个或多个索引方法特定的存储参数。 需要注意的是索引内容不会被这个命令立即修改，根据参数的不同，可能需要使用REINDEX重建索引来获得期望的效果。

- **RESET ({ storage_parameter } [, ...])**

  重置索引的一个或多个索引方法特定的存储参数为缺省值。与SET一样，可能需要使用REINDEX来完全更新索引。

- **[MODIFY PARTITION index_partition_name] UNUSABLE**

  用于设置表或者索引分区上的索引不可用。

- **REBUILD [PARTITION index_partition_name]**

  用于重建表或者索引分区上的索引。

- **RENAME PARTITION**

  用于重命名索引分区。

- **COMMENT comment_text**

  用于添加，修改或删除索引的注释。

- **index_name**
  要修改的索引名。
- **new_name**
  新的索引名。新的索引名不能与数据库中已有的表名重复。
  取值范围：字符串，且符合标识符命名规范。
- **storage_parameter**
  索引方法特定的参数名。
- **value**
  索引方法特定的存储参数的新值。根据参数的不同，这可能是一个数字或单词。
- **new_index_partition_name**
  新索引分区名。
- **index_partition_name**
  索引分区名。
- **comment_text**
  索引的注释信息。

示例

- 修改表的索引。

  创建示例表tpcds.ship_mode_t1：

```
DROP TABLE IF EXISTS tpcds.ship_mode_t1;
CREATE TABLE tpcds.ship_mode_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
)
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);
```

  在表tpcds.ship_mode_t1上的SM_SHIP_MODE_SK字段上创建唯一索引：

```
DROP INDEX IF EXISTS ds_ship_mode_t1_index1;
CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
```

  在表tpcds.ship_mode_t1上SM_CODE字段上创建表达式索引。

```
DROP INDEX IF EXISTS ds_ship_mode_t1_index2;
CREATE INDEX ds_ship_mode_t1_index2 ON tpcds.ship_mode_t1(SUBSTR(SM_CODE,1 ,4));
```

  重命名现有的索引ds_ship_mode_t1_index1为ds_ship_mode_t1_index5：

```
ALTER INDEX tpcds.ds_ship_mode_t1_index1 RENAME TO ds_ship_mode_t1_index5;
```

  设置索引ds_ship_mode_t1_index2不可用：

```
ALTER INDEX tpcds.ds_ship_mode_t1_index2 UNUSABLE;
```

  重建索引ds_ship_mode_t1_index2：

```
ALTER INDEX tpcds.ds_ship_mode_t1_index2 REBUILD;
```

- 修改表分区索引。

  创建示例表tpcds.customer_address_p1。

```
DROP TABLE IF EXISTS tpcds.customer_address_p1;
CREATE TABLE tpcds.customer_address_p1
(
    CA_ADDRESS_SK             INTEGER               NOT NULL,
    CA_ADDRESS_ID             CHAR(16)              NOT NULL,
```



```
    CA_STREET_NUMBER          CHAR(10)                  ,
    CA_STREET_NAME            VARCHAR(60)               ,
    CA_STREET_TYPE            CHAR(15)                  ,
    CA_SUITE_NUMBER           CHAR(10)                  ,
    CA_CITY                   VARCHAR(60)               ,
    CA_COUNTY                 VARCHAR(30)               ,
    CA_STATE                  CHAR(2)                   ,
    CA_ZIP                    CHAR(10)                  ,
    CA_COUNTRY                VARCHAR(20)               ,
    CA_GMT_OFFSET             DECIMAL(5,2)              ,
    CA_LOCATION_TYPE          CHAR(20)
)
DISTRIBUTE BY HASH(CA_ADDRESS_SK)
PARTITION BY RANGE(CA_ADDRESS_SK)
(
    PARTITION p1 VALUES LESS THAN (3000),
    PARTITION p2 VALUES LESS THAN (5000) ,
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

创建分区表索引ds_customer_address_p1_index2，并指定索引分区的名字。

```
DROP INDEX IF EXISTS ds_customer_address_p1_index2;
CREATE INDEX ds_customer_address_p1_index2 ON tpcds.customer_address_p1(CA_ADDRESS_SK)
LOCAL
(
    PARTITION CA_ADDRESS_SK_index1,
    PARTITION CA_ADDRESS_SK_index2,
    PARTITION CA_ADDRESS_SK_index3
)
;
```

重命名分区表的分区索引tpcds.ds_customer_address_p1_index2：

```
ALTER INDEX tpcds.ds_customer_address_p1_index2 RENAME PARTITION CA_ADDRESS_SK_index1 TO
CA_ADDRESS_SK_index4;
```

修改索引的注释：

```
ALTER INDEX tpcds.ds_customer_address_p1_index2 COMMENT
'comment_ds_customer_address_p1_index2';
```

相关链接

CREATE INDEX，**DROP INDEX**，**REINDEX**

12.9 ALTER LARGE OBJECT

功能描述

ALTER LARGE OBJECT用于更改一个large object的定义。它的唯一的功能是分配一个新的所有者。

注意事项

只有large object的所有者有权限执行此命令，系统管理员默认拥有此权限。

语法格式

```
ALTER LARGE OBJECT large_object_oid
    OWNER TO new_owner;
```

参数说明

- **large_object_oid**
  要被变large object的OID 。
  取值范围：已存在的大对象名。
- **OWNER TO new_owner**
  large object新的所有者。
  取值范围：已存在的用户名/角色名。

示例

无。

12.10 ALTER REDACTION POLICY

功能描述

修改应用在指定表的脱敏策略。

注意事项

只有待修改脱敏策略所应用表对象的属主才有修改脱敏策略的权限。

语法格式

- 修改脱敏策略生效表达式。
  ```
  ALTER REDACTION POLICY policy_name ON table_name WHEN (new_when_expression);
  ```
- 使脱敏策略生效或失效。
  ```
  ALTER REDACTION POLICY policy_name ON table_name ENABLE | DISABLE;
  ```
- 重命名脱敏策略。
  ```
  ALTER REDACTION POLICY policy_name ON table_name RENAME TO new_policy_name;
  ```
- 修改脱敏列，包括新增、修改、删除脱敏列。
  ```
  ALTER REDACTION POLICY policy_name ON table_name
    action;
  ```
  其中，脱敏列操作action可以是以下子句之一：
  ```
  ADD COLUMN column_name WITH function_name ( arguments )
   | MODIFY COLUMN column_name WITH function_name ( arguments )
   | DROP COLUMN column_name
  ```

参数说明

- **policy_name**
  待修改的脱敏策略名。
- **table_name**
  待修改的脱敏策略应用的表对象名。
- **new_when_expression**
  脱敏策略新的生效表达式。
- **ENABLE | DISABLE**
  当前脱敏策略是否生效。

  - ENABLE
    使之前失效的表对象的脱敏策略重新生效。
  - DISABLE
    使当前应用在表对象的脱敏策略失效。
- **new_policy_name**
  新的脱敏策略名。
- **column_name**
  脱敏表对象的列字段名。
  如果新增脱敏列，则指定列字段尚未绑定任何脱敏函数。
  如果修改脱敏列，则指定列字段为已存在的脱敏列。
  如果删除脱敏列，则指定列字段为已存在的脱敏列。
- **function_name**
  脱敏函数名。
- **arguments**
  脱敏函数的参数列表。
  - MASK_NONE，表示不进行任何脱敏处理。
  - MASK_FULL，表示全脱敏成固定值。
  - MASK_PARTIAL，表示按指定的字符类型，数值类型或时间类型进行部分脱敏处理。

示例

创建用户test_role和示例表emp并插入数据：

```
CREATE ROLE test_role PASSWORD '{Password}';

DROP TABLE IF EXISTS emp;
CREATE TABLE emp(id int, name varchar(20), salary NUMERIC(10,2));
INSERT INTO emp VALUES(1, 'July', 1230.10), (2, 'David', 999.99);
```

为表对象emp创建脱敏策略mask_emp，字段salary对用户test_role不可见：

```
CREATE REDACTION POLICY mask_emp ON emp WHEN(current_user = 'test_role') ADD COLUMN salary
WITH mask_full(salary);
```

修改脱敏策略生效表达式，使其对指定角色生效（若不指定用户，默认对当前用户生效）：

```
ALTER REDACTION POLICY mask_emp ON emp WHEN (pg_has_role(current_user, 'redact_role', 'member'));
ALTER REDACTION POLICY mask_emp ON emp WHEN (pg_has_role('redact_role', 'member'));
```

修改脱敏策略生效表达式，使其对所有用户均生效：

```
ALTER REDACTION POLICY mask_emp ON emp WHEN (1=1);
```

修改脱敏策略，使其失效：

```
ALTER REDACTION POLICY mask_emp ON emp DISABLE;
```

重新使脱敏策略生效：

```
ALTER REDACTION POLICY mask_emp ON emp ENABLE;
```

重命名脱敏策略为mask_emp_new：

```
ALTER REDACTION POLICY mask_emp ON emp RENAME TO mask_emp_new;
```

新增脱敏列：


```
ALTER REDACTION POLICY mask_emp_new ON emp ADD COLUMN name WITH mask_partial(name, '*', 1,
length(name));
```

修改脱敏列name，采用脱敏函数MASK_FULL对name字段数据全脱敏：

```
ALTER REDACTION POLICY mask_emp_new ON emp MODIFY COLUMN name WITH mask_full(name);
```

删除已存在的脱敏列：

```
ALTER REDACTION POLICY mask_emp_new ON emp DROP COLUMN name;
```

相关链接

CREATE REDACTION POLICY，**DROP REDACTION POLICY**

12.11 ALTER RESOURCE POOL

功能描述

修改一个资源池，指定其他控制组。

注意事项

只要用户对当前数据库有ALTER权限，就可以修改资源池。

语法格式

```
ALTER RESOURCE POOL pool_name
    WITH ({MEM_PERCENT= pct | CONTROL_GROUP="group_name" | ACTIVE_STATEMENTS=stmt |
MAX_DOP = dop | MEMORY_LIMIT='memory_size' | io_limits=io_limits | io_priority='io_priority'}[, ... ]);
```

参数说明

- **pool_name**

  资源池名称。

  资源池名称为已创建的资源池。

  取值范围：字符串，要符合标识符的命名规范。

- **group_name**

  控制组名称。

  说明

  - 设置控制组名称时，语法可以使用双引号，也可以使用单引号。
  - group_name对大小写敏感。
  - 不指定group_name时，默认指定的字符串为 "Medium"，代表指定DefaultClass控制组的

    "Medium" Timeshare控制组。
  - 若数据库用户指定Timeshare控制组代表的字符串，即"Rush"、"High"、"Medium"或"Low"其中一种，如control_group的字符串为"High"；代表资源池指定到DefaultClass控制组下的"High" Timeshare控制组。

  取值范围：已创建的控制组。

- **stmt**

  资源池语句执行的最大并发数量。

取值范围：数值型，-1~INT_MAX。

- **dop**
  保留字段。
  取值范围：数值型，1~INT_MAX。
- **memory_size**
  资源池最大使用内存。
  取值范围：字符串，内容范围1KB~2047GB。
- **mem_percent**
  资源池可用内存占全部内存或者组用户内存使用的比例。
  普通用户的mem_percent范围为0-100的整数，默认值为0。
- **io_limits**
  该参数8.1.2集群版本中已废弃，为兼容历史版本保留该参数。
- **io_priority**
  该参数8.1.2集群版本中已废弃，为兼容历史版本保留该参数。

示例

创建示例资源池pool_test，其控制组为“DefaultClass”组下属的“Medium” Timeshare Workload控制组：

```
CREATE RESOURCE POOL pool_test;
```

修改资源池pool_test，其控制组指定为"DefaultClass"组下属的"High" Timeshare Workload控制组：

```
ALTER RESOURCE POOL pool_test WITH (CONTROL_GROUP="High");
```

相关链接

CREATE RESOURCE POOL，**DROP RESOURCE POOL**

12.12 ALTER ROLE

功能描述

修改角色属性。

注意事项

无。

语法格式

- 修改角色的权限。
  ```
  ALTER ROLE role_name [ [ WITH ] option [ ... ] ];
  ```
  其中权限项子句option为。
  ```
  {CREATEDB | NOCREATEDB}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {AUDITADMIN | NOAUDITADMIN}
  ```

```
    | {SYSADMIN | NOSYSADMIN}
    | {USEFT | NOUSEFT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY 'password' [ REPLACE 'old_password' ]
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' | DISABLE }
    | [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE 'old_password' ] |
DISABLE }
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_cluster_name
    | ACCOUNT { LOCK | UNLOCK }
    | PGUSER
    | AUTHINFO 'authinfo'
    | PASSWORD EXPIRATION period
```

- 修改角色的名字。

```
ALTER ROLE role_name
    RENAME TO new_name;
```

- 设置角色的配置参数。

```
ALTER ROLE role_name [ IN DATABASE database_name ]
    SET configuration_parameter {{ TO | = } { value | DEFAULT } | FROM CURRENT};
```

- 重置角色的配置参数。

```
ALTER ROLE role_name
    [ IN DATABASE database_name ] RESET {configuration_parameter|ALL};
```

参数说明

- **role_name**

  现有角色名。

  取值范围：已存在的用户名。

- **new_name**

  角色的新名称。

  取值范围：字符串，要符合标识符的命名规范。且最多为63个字符。

- **CREATEDB | NOCREATEDB**

  决定一个新角色是否能创建数据库。

  新角色没有创建数据库的权限。

  缺省为NOCREATEDB。

- **CREATEROLE | NOCREATEROLE**

  决定一个角色是否可以创建新角色（也就是执行CREATE ROLE和CREATE USER）。一个拥有CREATEROLE权限的角色也可以修改和删除其他角色。

  缺省为NOCREATEROLE。

- **INHERIT | NOINHERIT**

  决定一个角色是否“继承”它所在组的角色的权限。不推荐使用。

- **AUDITADMIN | NOAUDITADMIN**

  定义角色是否有审计管理属性。

缺省为NOAUDITADMIN。

- **SYSADMIN | NOSYSADMIN**

  决定一个新角色是否为“系统管理员”，具有SYSADMIN属性的角色拥有系统最高权限。

  缺省为NOSYSADMIN。

- **USEFT | NOUSEFT**

  决定一个新角色是否能操作外表，包括：新建外表、删除外表、修改外表、读写外表。

  新角色没有操作外表的权限。

  缺省为NOUSEFT，表示新建角色默认不能操作外表。

- **LOGIN | NOLOGIN**

  具有LOGIN属性的角色才可以登录数据库。一个拥有LOGIN属性的角色可以认为是一个用户。

  缺省为NOLOGIN。

- **REPLICATION | NOREPLICATION**

  定义角色是否允许流复制或设置系统为备份模式。REPLICATION属性是特定的角色，仅用于复制。

  缺省为NOREPLICATION。

- **INDEPENDENT | NOINDEPENDENT**

  定义私有、独立的角色。具有INDEPENDENT属性的角色，管理员对其进行的控制、访问的权限被分离，具体规则如下：

  - 未经INDEPENDENT角色授权，管理员无权对其表对象进行增、删、查、改、复制、授权操作。
  - 未经INDEPENDENT角色授权，管理员无权修改INDEPENDENT角色的继承关系。
  - 管理员无权修改INDEPENDENT角色的表对象的属主。
  - 管理员无权去除INDEPENDENT角色的INDEPENDENT属性。
  - 管理员无权修改INDEPENDENT角色的数据库密码，INDEPENDENT角色需管理好自身密码，密码丢失无法重置。
  - 管理员属性用户不允许定义修改为INDEPENDENT属性。

- **VCADMIN | NOVCADMIN**

  定义逻辑集群管理员角色。具有逻辑集群管理员属性的角色，和普通用户相比，有如下额外权限：

  - 在所关联逻辑集群中创建、修改和删除资源池的权限。
  - 将所关联的逻辑集群的访问权限授予其他用户或角色，或回收其他用户或角色对关联逻辑集群的访问权限。

- **CONNECTION LIMIT**

  声明该角色可以在单个CN上使用的并发连接数量。

  取值范围：整数，>=-1，缺省值为-1，表示没有限制。

须知

为保证集群正常使用，connection limit的最小值是集群中CN的数目。在集群做ANALYZE时，其他CN节点会连接当前做ANALYZE的CN节点来同步元数据。例如集群中有3个CN节点，那么connection limit应该设置为>=3。

- **ENCRYPTED | UNENCRYPTED**

  控制密码存储在系统表里的密码是否加密。（如果没有指定，那么缺省的行为由配置参数password_encryption_type控制。）按照产品安全要求，密码必须加密存储，所以，UNENCRYPTED在GaussDB(DWS)中禁止使用。因为系统无法对指定的加密密码字符串进行解密，所以如果目前的密码字符串已经是用SHA256加密的格式，则会继续照此存放，而不管是否声明了ENCRYPTED或UNENCRYPTED。这样就允许在dump/restore的时候重新加载加密的密码。

  – password

    登录密码。

    密码规则：密码默认不少于8个字符；不能与用户名及用户名倒序相同；至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。使用范围外的字符会收到告警，但依然允许创建。

    取值范围：字符串。

  – DISABLE

    默认情况下，用户可以更改自己的密码，除非密码被禁用。要禁用用户的密码，请指定DISABLE。禁用某个用户的密码后，将从系统中删除该密码，此类用户只能通过外部认证来连接数据库，例如：IAM认证、kerberos或ldap认证。只有管理员才能启用或禁用密码。普通用户不能禁用初始用户的密码。要启用密码，请运行ALTER USER并指定密码。

- **VALID BEGIN**

  设置角色生效的时间戳。如果省略了该子句，角色无有效开始时间限制。

- **VALID UNTIL**

  设置角色失效的时间戳。如果省略了该子句，角色无有效结束时间限制。

- **RESOURCE POOL**

  设置角色使用的resource pool名字，该名字属于系统表：pg_resource_pool

- **USER GROUP 'groupuser'**

  创建一个user的子用户。

- **PERM SPACE**

  设置用户永久表存储空间限额。

  space_limit：永久表存储空间上限。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **TEMP SPACE**

  设置用户临时表存储空间限额。

  tmpspacelimit：临时表存储空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **SPILL SPACE**

  设置用户算子落盘空间限额。

spillspacelimit：算子落盘空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **NODE GROUP**

  设置用户关联的逻辑集群名称。如果需要关联的逻辑集群名称包含大写字符或特殊字符，指定逻辑集群名称时需要加双引号。

- **ACCOUNT LOCK | ACCOUNT UNLOCK**
  - ACCOUNT LOCK：锁定账户，禁止登录数据库。
  - ACCOUNT UNLOCK：解锁账户，允许登录数据库。

- **PGUSER**

  该属性用于兼容开源Postgres的连接通讯，开源的Postgres客户端接口（推荐使用Postgres 9.2.19版本的相关客户端接口）可以使用具有该属性的数据库用户连接数据库。

  当前版本不允许修改角色的PGUSER属性。

须知

该属性只用于兼容连接过程，而由于本产品与Postgres的内核差异导致的不兼容，不在此属性控制范围内。

由于具有PGUSER属性的用户的认证方式与其他用户不同，开源客户端的报错信息可能导致数据库用户PGUSER属性被枚举，建议使用本产品自有的客户端。例如：

```
#normaluser是不具有PGUSER属性的用户，psql是Postgres的客户端工具
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser
psql: authentication method 10 not supported

#pguser用户是具有PGUSER属性的用户
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser
Password for user pguser:
```

- **AUTHINFO 'authinfo'**

  该属性用于指定角色认证类型，authinfo为类型说明字符串，大小写敏感。当前仅支持LDAP类型，对应的类型说明字符串为'ldap'。LDAP属于外部认证，故需要同时指定PASSWORD DISABLE。

须知

- authinfo可以加上LDAP认证的额外信息，比如LDAP认证中的fulluser，fulluser等同于ldapprefix+username+ldapsuffix。当authinfo为'ldap'，表示角色认证类型为LDAP，此时ldapprefix和ldapsuffix的信息由pg_hba.conf中匹配的记录提供。
- ALTER ROLE时不允许用户切换认证类型，仅允许LDAP用户修改LDAP属性。

- **PASSWORD EXPIRATION period**

  声明该角色的登录密码过期天数，登录密码过期之前用户需要及时修改密码。登录密码过期后用户无法登录，需要请管理员设置新的登录密码后登录。

  取值范围：整数，-1~999。缺省值为-1，表示没有过期限制；0表示用户登录密码立即过期。

- **IN DATABASE database_name**

  表示修改角色在指定数据库上的参数。

- **SET configuration_parameter**

  设置角色的参数。ALTER ROLE中修改的会话参数只针对指定的角色，且在下一次该角色启动的会话中有效。

  取值范围：

  configuration_parameter和value的取值请参见**SET**。

  DEFAULT：表示清除configuration_parameter参数的值，configuration_parameter参数的值将继承本角色新产生的SESSION的默认值。

  FROM CURRENT：取当前会话中的值设置为configuration_parameter参数的值。

- **RESET configuration_parameter|ALL**

  清除configuration_parameter参数的值。与SET configuration_parameter TO DEFAULT的效果相同。

  取值范围：ALL表示清除所有参数的值。

示例

创建示例角色r1，r2和r3。

```
CREATE ROLE r1 IDENTIFIED BY '{Password}';
CREATE ROLE r2 WITH LOGIN AUTHINFO 'ldapcn=r2,cn=user,dc=lework,dc=com' PASSWORD DISABLE;
CREATE ROLE r3 WITH LOGIN PASSWORD '{Password}' PASSWORD EXPIRATION 30;
```

修改角色r1的登录权限：

```
ALTER ROLE r1 login;
```

修改角色r1的密码：

```
ALTER ROLE r1 IDENTIFIED BY '{new_Password}' REPLACE '{Password}';
```

修改角色manager为系统管理员：

```
ALTER ROLE r1 SYSADMIN;
```

修改LDAP认证角色的fulluser信息：

```
ALTER ROLE r2 WITH LOGIN AUTHINFO 'ldapcn=role2,cn=user2,dc=func,dc=com' PASSWORD DISABLE;
```

修改角色的登录密码有效期为90天：

```
ALTER ROLE r3 PASSWORD EXPIRATION 90;
```

相关链接

CREATE ROLE，**DROP ROLE**，**SET**

12.13 ALTER ROW LEVEL SECURITY POLICY

功能描述

对已存在的行访问控制策略（包括行访问控制策略的名称，行访问控制指定的用户，行访问控制的策略表达式）进行修改。

注意事项

表的所有者或管理员用户才能进行此操作。

语法格式

```
ALTER [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name RENAME TO
new_policy_name

ALTER [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name
    [ TO { role_name | PUBLIC } [, ...] ]
    [ USING ( using_expression ) ]
```

参数说明

- **policy_name**

  行访问控制策略名称。

- **table_name**

  行访问控制策略的表名。

- **new_policy_name**

  新的行访问控制策略名称。

- **role_name**

  行访问控制策略应用的数据库用户，可以指定多个用户，PUBLIC表示应用到所有用户。

- **using_expression**

  行访问控制的表达式，返回值为boolean类型。

示例

创建示例用户role_a和role_b：

```
CREATE ROLE role_a PASSWORD '{Password}';
CREATE ROLE role_b PASSWORD '{Password}';
```

创建示例数据表public.all_data_t并插入数据：

```
CREATE TABLE public.all_data_t(id int, role varchar(100), data varchar(100));
INSERT INTO all_data_t VALUES(1, 'role_a', 'r_a_data');
INSERT INTO all_data_t VALUES(2, 'role_b', 'r_b_data');
INSERT INTO all_data_t VALUES(3, 'role_c', 'r_c_data');
```

创建示例行访问控制策略：

```
CREATE ROW LEVEL SECURITY POLICY all_data_t_rls ON all_data_t USING(role = CURRENT_USER);
```

打开行访问控制策略开关：

```
ALTER TABLE all_data_t ENABLE ROW LEVEL SECURITY;
```

修改行访问控制all_data_rls的名称：

```
ALTER ROW LEVEL SECURITY POLICY all_data_t_rls ON all_data_t RENAME TO all_data_t_newrls;
```

修改行访问控制策略影响的用户：

```
ALTER ROW LEVEL SECURITY POLICY all_data_t_newrls ON all_data_t TO role_a, role_b;
```

修改行访问控制策略表达式：

```
ALTER ROW LEVEL SECURITY POLICY all_data_t_newrls ON all_data_t USING (id > 100 AND role =
current_user);
```


相关链接

CREATE ROW LEVEL SECURITY POLICY，DROP ROW LEVEL SECURITY POLICY

12.14 ALTER SCHEMA

功能描述

修改模式属性。

注意事项

只有模式的所有者或者被授予了模式ALTER权限的用户，有权限执行ALTER SCHEMA命令，系统管理员默认拥有此权限。

若要修改模式的所有者，当前用户必须是该模式的所有者或者系统管理员。

语法格式

- 修改模式的名称。
```
ALTER SCHEMA schema_name
    RENAME TO new_name;
```
- 修改模式的所有者。
```
ALTER SCHEMA schema_name
    OWNER TO new_owner;
```
- 修改模式的永久表存储空间限制。
```
ALTER SCHEMA schema_name
    WITH PERM SPACE 'space_limit';
```

参数说明

- **schema_name**

  现有模式的名字。

  取值范围：已存在的模式名。
- **RENAME TO new_name**

  修改模式的名字。

  new_name：模式的新名字。

  取值范围：字符串，要符合标识符命名规范。
- **OWNER TO new_owner**

  修改模式的所有者。非系统管理员要改变模式的所有者，该用户还必须是新的所有角色的直接或间接成员， 并且该成员必须在此数据库上有CREATE权限。

  new_owner：模式的新所有者。

  取值范围：已存在的用户名/角色名。
- **WITH PERM SPACE**

  修改模式的永久表存储空间上限。非系统管理员要改变模式的存储空间上限，该用户还必须是新的所有角色的直接或间接成员， 并且该成员必须在此数据库上有CREATE权限。

  space_limit：新的模式永久表存储空间上限。

取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。解析后的数值以K为单位，且范围不能够超过64比特表示的有符号整数，即1KB~9007199254740991KB。

示例

创建示例模式schema_test和用户user_a：

```
CREATE SCHEMA schema_test;
CREATE USER user_a PASSWORD '{Password}';
```

将当前模式schema_test更名为schema_test1：

```
ALTER SCHEMA schema_test RENAME TO schema_test1;
```

将schema_test1的所有者修改为user_a：

```
ALTER SCHEMA schema_test1 OWNER TO user_a;
```

相关链接

CREATE SCHEMA，**DROP SCHEMA**

12.15 ALTER SEQUENCE

功能描述

修改序列定义。

注意事项

- 使用ALTER SEQUENCE的用户必须是该序列的所有者。
- 当前版本仅支持修改拥有者、归属列和最大值。若要修改其他参数，可以删除重建，并用Setval函数恢复当前值。
- ALTER SEQUENCE MAXVALUE不支持在事务、函数和存储过程中使用。
- 修改序列的最大值后，会清空该序列在所有会话的cache。
- ALTER SEQUENCE会阻塞nextval、setval、currval和lastval的调用。

语法格式

修改序列最大值或归属列

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ MAXVALUE maxvalue | NO MAXVALUE | NOMAXVALUE ]
    [ OWNED BY { table_name.column_name | NONE } ] ;
```

修改序列的拥有者

```
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner;
```

参数说明

- **name**

  要修改的序列名称。

- **IF EXISTS**

  当序列不存在时使用该选项不会出现错误消息，仅有一个通知。

- **MAXVALUE maxvalue | NO MAXVALUE**

  序列所能达到的最大值。如果声明了NO MAXVALUE，则递增序列的缺省值为2^{63}-1，递减序列的缺省值为-1。NOMAXVALUE等价于NO MAXVALUE。

- **OWNED BY**

  将序列和一个表的指定字段进行关联。这样，在删除指定字段或其所在表的时候会自动删除已关联的序列。

  如果序列已经和表有关联后，使用OWNED BY参数后新的关联关系会覆盖旧的关联。

  关联的表和序列的所有者必须是同一个用户，并且在同一个模式中。

  使用OWNED BY NONE将删除任何已经存在的关联。

- **new_owner**

  序列新所有者的用户名。用户要修改序列的所有者，必须是新角色的直接或者间接成员，并且所有者角色必须有序列所在模式上的CREATE权限。

示例

创建示例序列seq_test和示例表t1：

```
CREATE SEQUENCE seq_test START 101;
DROP TABLE IF EXISTS t1;
CREATE TABLE t1(c1 bigint default nextval('seq_test'));
```

将序列serial的最大值修改为200：

```
ALTER SEQUENCE seq_test MAXVALUE 200;
```

将序列seq_test的归属列变为t1.c1：

```
ALTER SEQUENCE seq_test OWNED BY t1.c1;
```

相关链接

CREATE SEQUENCE，**DROP SEQUENCE**

12.16 ALTER SERVER

功能描述

增加、修改和删除外部服务器的定义。

现有外部服务器（server）可以从pg_foreign_server系统表查询。

注意事项

除系统管理员外，只有server的owner才可以进行ALTER操作。

语法格式

- 修改外部服务的参数。

```
ALTER SERVER server_name [ VERSION 'new_version' ]
    [ OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] ) ];
```

在OPTIONS选项里，ADD、SET和DROP指定要执行的操作，未指定时默认为ADD操作。option和value为对应操作的参数。

对于HDFS Server目前只支持SET操作，ADD/DROP操作现有版本不支持。语法中SET和DROP操作语法依然保留，以便后续扩展使用。

- 修改外部服务的所有者。

```
ALTER SERVER server_name
    OWNER TO new_owner;
```

- 修改外部服务的名字。

```
ALTER SERVER server_name
    RENAME TO new_name;
```

- 刷新HDFS配置文件。

```
ALTER SERVER server_name REFRESH OPTIONS;
```

参数说明

- **server_name**

  要修改的server的名字。

- **new_version**

  修改后server的新版本名称。

- **OPTIONS**：
  - address

    OBS服务的终端节点（Endpoint）。

    HDFS集群的主备节点所在的IP地址以及端口。

    说明

    - 对于HDFS server，address必须存在，所以ADD和DROP操作不被允许。
    - address目前只支持点分十进制格式的ipv4格式，且address字符串中不能出现空格，多组address以逗号作为分隔符。ip和port之间使用“:”来区分。HDFS集群中ip、port组对推荐设置两组，分别对应HDFS NameNode主备节点的address。
    - 当server类型为DLI时，address为DLI服务上数据所存储的OBS address。
  - hdfscfgpath

    HDFS集群的配置文件。

    说明

    - 若HDFS走安全模式时，hdfscfgpath是必选项，否则为可选项。
    - 若设置hdfscfgpath时，path仅能设置一个。
  - fed

    表示dfs_fdw连接的是HDFS为联邦模式。

    取值rbf，表示HDFS为联邦rbf方式。

    说明

    该参数8.1.2及以上版本支持；8.0.0基线版本下，仅8.0.0.10及以上版本支持。
  - encrypt

    是否对数据进行加密，该参数仅支持在type为OBS时设置。默认值为off。

取值范围：

- on表示对数据进行加密。
- off表示不对数据进行加密。

– access_key

OBS访问协议对应的AK值（OBS云服务界面由用户获取），创建外表时AK值会加密保存到数据库的元数据表中。该参数仅支持type为OBS时设置。

– secret_access_key

OBS访问协议对应的SK值（OBS云服务界面由用户获取），创建外表时SK值会加密保存到数据库的元数据表中。该参数仅支持type为OBS时设置。

– dli_address

DLI服务的终端节点，即endpoint。该参数仅支持type为DLI时设置。

– dli_access_key

DLI访问协议对应的AK值（DLI云服务界面由用户获取），创建外表时AK值会保存到数据库的元数据表中。该参数仅支持type为DLI时设置。

– dli_secret_access_key

DLI访问协议对应的SK值（DLI云服务界面由用户获取），创建外表时SK值会加密保存到数据库的元数据表中。该参数仅支持type为DLI时设置。

– region

此参数表示OBS服务的IP地址或者域名信息。该参数仅支持type为OBS时设置。

– dbname

用于协同分析、跨集群互联互通，表示将要连接的远端集群的数据库名字。

– username

用于协同分析、跨集群互联互通，表示将要连接的远端集群的用户名。

– password

用于协同分析、跨集群互联互通，表示将要连接的远端集群的用户名密码。

– syncsrv

仅用于跨集群互联互通，表示数据同步过程中使用到的GDS服务，设置方式与GDS外表的location属性相同。该参数仅8.1.2及以上版本支持。

- **new_owner**

  修改后server的新所有者。更改所有者，必须是外部服务器的所有者并且也是新的所有者角色的直接或者间接成员，并且必须对外部服务器的外部数据封装器有USAGE权限。

- **new_name**

  修改后server的新名称。

- **REFRESH OPTIONS**

  刷新HDFS的配置文件信息，在配置文件有变动时执行，若不执行可能会有访问报错。

示例

建立一个hdfs_server，其中hdfs_fdw为数据库中存在的foreign data wrapper：

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS
  (address '10.10.0.100:25000,10.10.0.101:25000',
```



```
    hdfscfgpath '/opt/hadoop_client/HDFS/hadoop/etc/hadoop',
    type 'HDFS'
) ;
```

修改现有名为hdfs_server的address：

```
ALTER SERVER hdfs_server OPTIONS ( SET address '10.10.0.110:25000,10.10.0.120:25000');
```

修改现有名为hdfs_server的hdfscfgpath：

```
ALTER SERVER hdfs_server OPTIONS ( SET hdfscfgpath '/opt/bigdata/hadoop');
```

相关链接

CREATE SERVER DROP SERVER

12.17 ALTER SESSION

功能描述

ALTER SESSION命令用于定义或修改那些对当前会话有影响的条件或参数。修改后的会话参数会一直保持，直到断开当前会话。

注意事项

- 如果执行SET TRANSACTION之前没有执行START TRANSACTION，则事务立即结束，命令无法显示效果。
- 可以用START TRANSACTION里面声明所需要的transaction_mode(s)的方法来避免使用SET TRANSACTION。

语法格式

- 设置会话的事务参数。

```
ALTER SESSION SET [ SESSION CHARACTERISTICS AS ] TRANSACTION
   { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED } | { READ ONLY  | READ
WRITE } } [, ...] ;
```

- 设置会话的其他运行时参数。

```
ALTER SESSION SET
   {{config_parameter { { TO  | =  }  { value | DEFAULT }
     | FROM CURRENT }} | CURRENT_SCHEMA [ TO | = ] { schema | DEFAULT }
     | TIME ZONE time_zone
     | SCHEMA schema
     | NAMES encoding_name
     | ROLE role_name PASSWORD 'password'
     | SESSION AUTHORIZATION { role_name PASSWORD 'password' | DEFAULT }
     | XML OPTION { DOCUMENT | CONTENT }
   } ;
```

参数说明

- **SESSION**

  声明的参数只对当前会话起作用。如果SESSION和LOCAL都没出现，则SESSION为缺省值。

  如果在事务中执行了此命令，命令的产生影响将在事务回滚之后消失。如果该事务已提交，影响将持续到会话的结束，除非被另外一个SET命令重置参数。

- **config_parameter**

可设置的运行时参数的名称。可用的运行时参数可以使用SHOW ALL命令查看。

说明

部分通过SHOW ALL查看的参数不能通过SET设置。如max_datanodes。

- **value**

  config_parameter的新值。可以声明为字符串常量、标识符、数字，或者逗号分隔的列表。DEFAULT用于把这些参数设置为它们的缺省值。

- **TIME ZONE timezone**

  用于指定当前会话的本地时区。

  取值范围：有效的本地时区。该选项对应的运行时参数名称为TimeZone，DEFAULT缺省值为PRC。

- **CURRENT_SCHEMA**

  CURRENT_SCHEMA用于指定当前的模式。

  取值范围：已存在模式名称。

- **SCHEMA schema**

  同CURRENT_SCHEMA。此处的schema是个字符串。

  例如：set schema 'public';

- **NAMES encoding_name**

  用于设置客户端的字符编码。等价于set client_encoding to encoding_name。

  取值范围：有效的字符编码。该选项对应的运行时参数名称为client_encoding，默认编码为UTF8。

- **XML OPTION option**

  用于设置XML的解析方式。

  取值范围：CONTENT（缺省），DOCUMENT。

示例

创建模式ds：

```
CREATE SCHEMA ds;
```

设置模式搜索路径：

```
SET SEARCH_PATH TO ds, public;
```

设置日期时间风格为传统的POSTGRES风格（日在月前）：

```
SET DATESTYLE TO postgres, dmy;
```

设置当前会话的字符编码为UTF8：

```
ALTER SESSION SET NAMES 'UTF8';
```

设置时区为加州伯克利：

```
SET TIME ZONE 'PST8PDT';
```

设置时区为意大利：

```
SET TIME ZONE 'Europe/Rome';
```

设置当前模式：

```
ALTER SESSION SET CURRENT_SCHEMA TO tpcds;
```


设置XML OPTION为DOCUMENT：

```
ALTER SESSION SET XML OPTION DOCUMENT;
```

创建角色joe，并设置会话的角色为joe：

```
CREATE ROLE joe WITH PASSWORD '{Password}';
ALTER SESSION SET SESSION AUTHORIZATION joe PASSWORD '{Password}';
```

切换到默认用户：

```
ALTER SESSION SET SESSION AUTHORIZATION default;
```

相关链接

SET

12.18 ALTER SYNONYM

功能描述

修改SYNONYM对象的属性。

注意事项

- 目前仅支持修改SYNONYM对象的属主。
- 只有系统管理员和同义词的属主有权限修改SYNONYM对象的属主信息。
- 修改者必须是新属主的直接或间接成员，并且新属主必须拥有该同义词所属模式的CREATE权限。

语法格式

```
ALTER SYNONYM synonym_name
    OWNER TO new_owner;
```

参数描述

- **synonym**

  待修改的同义词名字，可以带模式名。

  取值范围：字符串，需要符合标识符的命名规范。

- **new_owner**

  同义词对象的新所有者。

  取值范围：字符串，有效的用户名。

示例

创建同义词t1：

```
CREATE OR REPLACE SYNONYM t1 FOR ot.t1;
```

创建新用户u1：

```
CREATE USER u1 PASSWORD '{Password}';
```

修改同义词t1的owner为u1：


```
ALTER SYNONYM t1 OWNER TO u1;
```

相关链接

CREATE SYNONYM，**DROP SYNONYM**

12.19 ALTER SYSTEM KILL SESSION

功能描述

ALTER SYSTEM KILL SESSION命令用于结束一个会话。

注意事项

无。

语法格式

```
ALTER SYSTEM KILL SESSION 'session_sid, serial' [ IMMEDIATE ];
```

参数说明

- **session_sid, serial**
  会话的SID和SERIAL（格式请参考示例）。
  取值范围：通过查看系统表V$SESSION可查看所有会话的SID和SERIAL。
- **IMMEDIATE**
  表明会话将在命令执行后立即结束。

示例

查询会话信息：

```
SELECT sid,serial#,username FROM V$SESSION;

      sid       | serial# | username
----------------+---------+----------
 140131075880720 |       0 |
 140131025549072 |       0 |
 140131073779472 |       0 |
 140131071678224 |       0 |
 140131125774096 |       0 |
 140131127875344 |       0 |
 140131113629456 |       0 |
 140131094742800 |       0 |
(8 rows)
```

结束SID为140131075880720的会话：

```
ALTER SYSTEM KILL SESSION '140131075880720,0' IMMEDIATE;
```

12.20 ALTER TABLE

功能描述

修改表，包括修改表的定义、重命名表、重命名表中指定的列、重命名表的约束、设置表的所属模式、添加/更新多个列、打开/关闭行访问控制开关。

注意事项

- 只有表的所有者或者被授予了表ALTER权限的用户有权限执行ALTER TABLE命令，系统管理员默认拥有此权限。若要修改表的所有者或者修改表的模式，当前用户必须是该表的所有者或者系统管理员。
- 不支持修改存储参数ORIENTATION。
- SET SCHEMA操作不支持修改为系统内部模式，当前仅支持用户模式之间的修改。
- 列存表支持PARTIAL CLUSTER KEY，不支持外键表级约束。列存表从8.1.1版本开始支持主键和唯一表级约束。
- 列存表只支持添加字段ADD COLUMN、修改字段的数据类型ALTER TYPE、设置单个字段的收集目标SET STATISTICS、支持更改表名字、支持删除字段DROP COLUMN。对于添加的字段和修改的字段类型要求是列存支持的**数据类型**。ALTER TYPE的USING选项只支持常量表达式和涉及本字段的表达式，暂不支持涉及其他字段的表达式。
- 列存表支持的字段约束包括NULL、NOT NULL和DEFAULT常量值；对字段约束的修改当前支持对DEFAULT值的修改（SET DEFAULT）、删除（DROP DEFAULT）和NOT NULL约束的删除；支持对非空约束NULL/NOT NULL的删除（DROP NOT NULL），暂不支持对非空约束NULL/NOT NULL的修改（SET NOT NULL）。
- 修改列存表存储参数COLVERSION或者enable_delta时，不能与其他ALTER操作同时进行。
- 不支持增加自增列，或者增加DEFAULT值中包含nextval()表达式的列。
- 不支持对HDFS表、外表、临时表开启行访问控制开关。
- 通过约束名删除PRIMARY KEY约束时，不会删除NOT NULL约束。如有需要，可手动删除NOT NULL约束。
- OBS冷热表不支持ALTER RESET的cold_tablespace，storage_policy参数，不支持修改COLVERSION为1.0。
- 可将原列存COLVERSION为2.0表ALTER为OBS冷热表，需同时增加cold_tablespace和storage_policy参数。
- ALTER TABLE支持针对RELOPTIONS的storage_policy的选项更改。冷热切换策略发生更改后，不改变现有冷数据的冷热属性，只改变下一次执行冷热切换的规则，下一次执行冷热切换命令时将按新规则进行冷热切换。
- 对表执行以下ALTER TABLE操作时会触发表重建（表重建过程中会先把数据转储到一个新的数据文件中，重建完成之后会删除原始文件），当表比较大时，重建会消耗较多的磁盘空间。当磁盘空间不足时，要谨慎对待大表ALTER TABLE操作，防止触发集群只读。
  - 修改列数据类型。

- 行存表增加列（包括oid列）。
- 列存表修改COLVERSION。
- 列存表增加列指定DEFAULT常量值，DEFAULT值中包含volatile函数以及DEFAULT值非NULL且不属于特定数据类型中的任何一个。

语法格式

- 修改表的定义。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }
    action [, ... ];
```

其中具体表操作action可以是以下子句之一：

```
column_clause
    | ADD table_constraint [ NOT VALID ]
    | ADD table_constraint_using_index
    | VALIDATE CONSTRAINT constraint_name
    | DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    | CLUSTER ON index_name
    | SET WITHOUT CLUSTER
    | SET ( {storage_parameter = value} [, ... ] )
    | RESET ( storage_parameter [, ... ] )
    | OWNER TO new_owner
    | SET TABLESPACE new_tablespace
    | SET {COMPRESS|NOCOMPRESS}
    | DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { HASH ( column_name [,...] ) } }
    | TO { GROUP groupname | NODE ( nodename [, ... ] ) }
    | ADD NODE ( nodename [, ... ] )
    | DELETE NODE ( nodename [, ... ] )
    | DISABLE TRIGGER [ trigger_name | ALL | USER ]
    | ENABLE TRIGGER [ trigger_name | ALL | USER ]
    | ENABLE REPLICA TRIGGER trigger_name
    | ENABLE ALWAYS TRIGGER trigger_name
    | DISABLE ROW LEVEL SECURITY
    | ENABLE ROW LEVEL SECURITY
    | FORCE ROW LEVEL SECURITY
    | NO FORCE ROW LEVEL SECURITY
    | REFRESH STORAGE
```


说明

- **ADD table_constraint [NOT VALID]**

  给表增加一个新的约束。

- **ADD table_constraint_using_index**

  根据已有唯一索引为表增加主键约束或唯一约束。

- **VALIDATE CONSTRAINT constraint_name**

  验证一个外键或是一个使用NOT VALID选项创建的检查类约束，通过扫描全表来保证所有记录都符合约束条件。如果约束已标记为有效时，什么操作也不会发生。

- **DROP CONSTRAINT [IF EXISTS] constraint_name [RESTRICT | CASCADE]**

  删除一个表上的约束。

- **CLUSTER ON index_name**

  为将来的CLUSTER操作选择默认索引。实际上并没有重新盘簇化处理该表。

- **SET WITHOUT CLUSTER**

  从表中删除最新使用的CLUSTER索引。这样会影响将来那些没有声明索引的集群操作。

- **SET ({storage_parameter = value} [, ...])**

  修改表的一个或多个存储参数。

- **RESET (storage_parameter [, ...])**

  重置表的一个或多个存储参数。与SET一样，根据参数的不同可能需要重写表才能获得想要的效果。

- **OWNER TO new_owner**

  将表、序列、视图的属主改变成指定的用户。

- **SET {COMPRESS|NOCOMPRESS}**

  修改表的压缩特性。表压缩特性的改变只会影响后续批量插入的数据的存储方式，对已有数据的存储毫无影响。也就是说，表压缩特性的修改会导致该表中同时存在着已压缩和未压缩的数据。

- **DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { HASH (column_name [,...]) } }**

  修改表的分布方式，在修改表分布信息的同时会将表数据在物理上按新分布方式重新分布，修改完成后建议对被修改表执行ANALYZE，以便收集全新的统计信息。

  说明

  - 本操作属于重大变更操作，涉及表分布信息的修改以及数据的物理重分布，修改过程中会阻塞业务，修改完成后原有业务的执行计划会发生变化，请按照正规变更流程进行。
  - 本操作属于资源密集操作，针对大表的分布方式修改，建议在计算和存储资源充裕情况下进行，保证整个集群和原表所在表空间有足够的剩余空间能存储一张与原表同等大小且按照新分布方式进行分布的表。

- **TO { GROUP groupname | NODE (nodename [, ...]) }**

  此语法仅在扩展模式（GUC参数support_extended_features为on时）下可用。该模式谨慎打开，主要供内部扩容工具使用，一般用户不应使用该模式。

- **ADD NODE (nodename [, ...])**

  此语法主要供内部扩容工具使用，一般用户不建议使用。

- **DELETE NODE (nodename [, ...])**

  此语法主要供内部缩容工具使用，一般用户不建议使用。

- **DISABLE TRIGGER [trigger_name | ALL | USER]**

  禁用trigger_name所表示的单个触发器，或禁用所有触发器，或仅禁用用户触发器（此选项不包括内部生成的约束触发器，例如，可延迟唯一性和排除约束的约束触发器）。

说明

应谨慎使用此功能，因为如果不执行触发器，则无法保证原先期望的约束的完整性。

- **ENABLE TRIGGER [trigger_name | ALL | USER]**

  启用trigger_name所表示的单个触发器，或启用所有触发器，或仅启用用户触发器。

- **ENABLE REPLICA TRIGGER trigger_name**

  触发器触发机制受配置变量session_replication_role的影响，当复制角色为“origin”（默认值）或“local”时，将触发简单启用的触发器。

  配置为ENABLE REPLICA的触发器仅在会话处于“replica”模式时触发。

- **ENABLE ALWAYS TRIGGER trigger_name**

  无论当前复制模式如何，配置为ENABLE ALWAYS的触发器都将触发。

- **DISABLE/ENABLE ROW LEVEL SECURITY**

  开启或关闭表的行访问控制开关。

  当开启行访问控制开关时，如果未在该数据表定义相关行访问控制策略，数据表的行级访问将不受影响；如果关闭表的行访问控制开关，即使定义了行访问控制策略，数据表的行访问也不受影响。详细信息参见**CREATE ROW LEVEL SECURITY POLICY**章节。

- **NO FORCE/FORCE ROW LEVEL SECURITY**

  强制开启或关闭表的行访问控制开关。

  默认情况，表所有者不受行访问控制特性影响，但当强制开启表的行访问控制开关时，表的所有者（不包含系统管理员用户）会受影响。系统管理员可以绕过所有的行访问控制策略，不受影响。

- **REFRESH STORAGE**

  根据OBS冷热表storage_policy所定义的规则，将符合条件的本地热分区切换为存储在OBS上的冷分区。

  例如创建OBS冷热表时，设置storage_policy 为 'LMT:10'，则在执行该操作时可将10日前无修改的分区切为冷存储，存至OBS中。

– 其中列相关的操作column_clause可以是以下子句之一：

```
ADD [ COLUMN ] column_name data_type [ compress_mode ] [ COLLATE collation ]
[ column_constraint [ ... ] ]
    | MODIFY [ COLUMN ] column_name data_type
    | MODIFY [ COLUMN ] column_name [ CONSTRAINT constraint_name ] NOT NULL
[ ENABLE ]
    | MODIFY [ COLUMN ] column_name [ CONSTRAINT constraint_name ] NULL
    | MODIFY [ COLUMN ] column_name DEFAULT default_expr
    | MODIFY [ COLUMN ] column_name COMMENT comment_text
    | DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    | ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
[ USING expression ]
    | ALTER [ COLUMN ] column_name { SET DEFAULT expression | DROP DEFAULT }
    | ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    | ALTER [ COLUMN ] column_name SET STATISTICS [PERCENT] integer
    | ADD STATISTICS (( column_1_name, column_2_name [, ...] ))
    | ADD { INDEX | UNIQUE [ INDEX ] } [ index_name ] ( { { column_name | ( expression ) }
[ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS LAST ] } [, ...] ) [ USING method ]
[ COMMENT 'text' ] LOCAL [ ( { PARTITION index_partition_name } [, ...] ) ] [ WITH
( { storage_parameter = value } [, ...] ) ]
    | ADD { INDEX | UNIQUE [ INDEX ] } [ index_name ] ({ { column_name | ( expression ) }
[ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] }[, ...] ) [ USING
method ] [ COMMENT 'text' ] [ WITH ( {storage_parameter = value} [, ... ] ) ] [ WHERE
predicate ]
    | DROP { INDEX | KEY } index_name
    | CHANGE [ COLUMN ] old_column_name new_column_name data_type [ [ CONSTRAINT
constraint_name ] NOT NULL [ ENABLE ] |
        [ CONSTRAINT constraint_name ] NULL | DEFAULT default_expr | COMMENT 'text' ]
    | DELETE STATISTICS (( column_1_name, column_2_name [, ...] ))
    | ALTER [ COLUMN ] column_name SET ( {attribute_option = value} [, ... ] )
    | ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    | ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
```

说明

- **ADD [COLUMN] column_name data_type [compress_mode] [COLLATE collation] [column_constraint [...]]**

  向表中增加一个新的字段。用ADD COLUMN增加一个字段，所有表中现有行都初始化为该字段的缺省值（如果没有声明DEFAULT子句，值为NULL）。

- **ADD ({ column_name data_type [compress_mode] } [, ...])**

  向表中增加多列。

- **MODIFY [COLUMN] column_name data_type**

  修改表已存在字段的数据类型。

- **MODIFY [COLUMN] column_name [CONSTRAINT constraint_name] NOT NULL [ENABLE]**

  为表的某列添加not null约束，列存表暂不支持。

- **MODIFY [COLUMN] column_name [CONSTRAINT constraint_name] NULL**

  为表的某列移除not null约束。

- **MODIFY [COLUMN] column_name DEFAULT default_expr**

  修改表的default值。

- **MODIFY [COLUMN] column_name COMMENT comment_text**

  修改表的注释信息。

- **DROP [COLUMN] [IF EXISTS] column_name [RESTRICT | CASCADE]**

  从表中删除一个字段，和这个字段相关的索引和表约束也会被自动删除。如果任何表之外的对象依赖于这个字段，必须声明CASCADE ，比如外键参考、视图等。

  DROP COLUMN命令并不是物理上把字段删除，而只是简单地把它标记为对SQL操作不可见。随后对该表的插入和更新将在该字段存储一个NULL。因此，删除一个字段是很快的，但是它不会立即释放表在磁盘上的空间，因为被删除了的字段占据的空间还没有回收。这些空间将在执行VACUUM时而得到回收。

- **ALTER [COLUMN] column_name [SET DATA] TYPE data_type [COLLATE collation] [USING expression]**

  改变表字段的数据类型，只允许相同大类的类型转换（数值之间，字符串之间，时间之间等）。该字段涉及的索引和简单的表约束将被自动地转换为使用新的字段类型，方法是重新分析最初提供的表达式。

  ALTER TYPE要求重写整个表的特性有时候是一个优点，因为重写的过程消除了表中没用的空间。比如，要想立刻回收被一个已经删除的字段占据的空间，最快的方法是

  ```
  ALTER TABLE table ALTER COLUMN anycol TYPE anytype;
  ```

  这里的anycol是任何在表中还存在的字段，而anytype是和该字段的原类型一样的类型。这样的结果是在表上没有任何可见的语意的变化，但是这个命令强制重写，这样就删除了不再使用的数据。

- **ALTER [COLUMN] column_name { SET DEFAULT expression | DROP DEFAULT }**

  为一个字段设置或者删除缺省值。请注意缺省值只应用于随后的INSERT命令，它们不会修改表中已经存在的行。也可以为视图创建缺省，这个时候它们是在视图的ON INSERT规则应用之前插入到INSERT句中的。

- **ALTER [COLUMN] column_name { SET | DROP } NOT NULL**

  修改一个字段是否允许NULL值或者拒绝NULL值。如果表在字段中包含非NULL，则只能使用SET NOT NULL。

- **ALTER [COLUMN] column_name SET STATISTICS [PERCENT] integer**

  为随后的ANALYZE操作设置针对每个字段的统计收集目标。目标的范围可以在0到10000之内设置。设置为-1时表示重新恢复到使用系统缺省的统计目标。

- **{ADD | DELETE} STATISTICS ((column_1_name, column_2_name [, ...]))**

  用于添加和删除多列统计信息声明（不实际进行多列统计信息收集），以便在后续进行全表或全库analyze时进行多列统计信息收集。每组多列统计信息最多支持32列。不支持添加/删除多列统计信息声明的表：系统表、外表。

- **ADD { INDEX | UNIQUE [INDEX] } [index_name] ({ { column_name | (expression) } [COLLATE collation] [opclass] [ASC | DESC] [NULLS LAST] } [, ...]) [USING method] [COMMENT 'text'] LOCAL [({ PARTITION index_partition_name } [, ...])] [WITH ({ storage_parameter = value } [, ...])]**

  为表的分区表创建索引，具体参数可参考**CREATE INDEX**。

- **ADD { INDEX | UNIQUE [INDEX] } [index_name] ({ { column_name | (expression) } [COLLATE collation] [opclass] [ASC | DESC] [NULLS { FIRST | LAST }] }[, ...]) [USING method] [COMMENT 'text'] [WITH ({storage_parameter = value} [, ...])] [WHERE predicate]**

  在表上创建索引，具体参数可参考**CREATE INDEX**。

- **DROP { INDEX | KEY } index_name**

  删除一个表上的索引。

- **CHANGE [COLUMN] old_column_name new_column_name data_type [[CONSTRAINT constraint_name] NOT NULL [ENABLE] |**

  [CONSTRAINT constraint_name] NULL | DEFAULT default_expr | COMMENT 'text']

  修改表中列信息，可将旧列名修改成新列名，以及修改列字段信息。

- **ALTER [COLUMN] column_name SET ({attribute_option = value} [, ...])**

  ALTER [COLUMN] column_name RESET (attribute_option [, ...])

  设置/重置属性选项。

  属性选项定义的参数有：n_distinct、n_distinct_inherited和cstore_cu_sample_ratio。n_distinct 设置并固定表的distinct值统计信息，n_distinct_inherited 设置并固定继承表的distinct值统计信息，cstore_cu_sample_ratio 设置对cstore列存表进行analyze时所选CU的比例。目前，禁止SET/RESET n_distinct_inherited参数。

  说明

  - n_distinct

    手动设置该列的distinct值统计信息。

    取值范围：-1.0 ~ INT_MAX

    默认值：0，表示不设置。

  - n_distinct_inherited

    手动设置继承表的该列的distinct值统计信息。

    取值范围：-1.0 ~ INT_MAX

    默认值：0，表示不设置。

  - cstore_cu_sample_ratio

    设置列存表执行analyze，计算需要采样的CU个数时，需要扩大的倍数。

    取值范围：1.0 ~ 10000.0

    默认值：1.0

- **ALTER [COLUMN] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }**

  为一个字段设置存储模式。这个设置控制这个字段是内联保存还是保存在一个附属的表里，以及数据是否要压缩。仅支持对行存表的设置；对列存表没有意义，执行时报错。SET STORAGE本身并不改变表上的任何东西，只是设置将来的表操作时，建议使用的策略。

  - 其中列约束column_constraint为：

```
[ CONSTRAINT constraint_name ]
  { NOT NULL |
    NULL |
    CHECK ( expression ) |
    DEFAULT default_expr  |
    UNIQUE index_parameters |
    PRIMARY KEY index_parameters }
  [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

  - 其中列的压缩可选项compress_mode为：

```
[ DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS ]
```

- 其中根据已有唯一索引为表增加主键约束或唯一约束table_constraint_using_index为：

```
[ CONSTRAINT constraint_name ]
  { UNIQUE | PRIMARY KEY } USING INDEX index_name
  [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

- 其中表约束table_constraint为：

```
[ CONSTRAINT constraint_name ]
  { CHECK ( expression ) |
    UNIQUE ( column_name [, ... ] ) index_parameters |
    PRIMARY KEY ( column_name [, ... ] ) index_parameters }

  [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

  其中索引参数index_parameters为：

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
  [ USING INDEX TABLESPACE tablespace_name ]
```

- 修改表已存在字段的数据类型，可同时修改空约束，default值和注释信息，只允许相同大类的类型转换（数值之间，字符串之间，时间之间等）。

```
ALTER TABLE [ IF EXISTS ] table_name
  MODIFY ( { column_name data_type | [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ] |
    [ CONSTRAINT constraint_name ] NULL | DEFAULT default_expr | COMMENT 'text' } [, ...] );
```

- 重命名表。对名字的修改不会影响所存储的数据。暂不支持新表名前带有原表的schema名。

```
ALTER TABLE [ IF EXISTS ] table_name
  RENAME TO new_table_name;
```

- 重命名表中指定的列。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name )}
  RENAME [ COLUMN ] column_name TO new_column_name;
```

- 重命名表的约束。

```
ALTER TABLE { table_name [*] | ONLY table_name | ONLY ( table_name ) }
  RENAME CONSTRAINT constraint_name TO new_constraint_name;
```

- 设置表的所属模式。

```
ALTER TABLE [ IF EXISTS ] table_name
  SET SCHEMA new_schema;
```


说明

- 这种形式把表移动到另外一个模式。相关的索引、约束都跟着移动。目前序列不支持改变schema。 若该表拥有序列，需要将序列删除，重建，或者取消拥有关系， 才能将表schema更改成功。
- 要修改一个表的模式，用户必须在新模式上拥有CREATE权限。要把该表添加为一个父表的新子表，用户必须同时又是父表的所有者。要修改所有者，用户还必须是新的所有角色的直接或间接成员，并且该成员必须在此表的模式上有CREATE权限。这些限制规定了该用户不能做出了重建和删除表之外的事情。不过，系统管理员可以以任何方式修改任意表的所有权限。
- 除了RENAME和SET SCHEMA之外所有动作都可以捆绑在一个经过多次修改的列表中并行使用。比如，可以在一个命令里增加几个字段或修改几个字段的类型。对于大表，此种操作带来的效率提升更明显，原因在于只需要对该大表做一次处理。
- 增加一个CHECK或NOT NULL约束将会扫描该表，以保证现有的行符合约束要求。
- 用一个非空缺省值增加一个字段或者改变一个字段的现有类型会重写整个表。对于大表来说，这个操作可能会花很长时间，并且它还临时需要两倍的磁盘空间。

- 添加多个列。

```
ALTER TABLE [ IF EXISTS ] table_name
    ADD ( { column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint [ ... ] ]} [, ...] );
```

- 更新多个列。

```
ALTER TABLE [ IF EXISTS ] table_name
    MODIFY ( { column_name data_type | column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ] | column_name [ CONSTRAINT constraint_name ] NULL } [, ...] );
```

参数说明

- **IF EXISTS**

  如果不存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表不存在。

- **table_name [*] | ONLY table_name | ONLY (table_name)**

  table_name是需要修改的表名。

  若声明了ONLY选项，则只有那个表被更改。若未声明ONLY，该表及其所有子表都将会被更改。另外，可以在表名称后面显示地增加*选项来指定包括子表，即表示所有后代表都被扫描，这是默认行为。

- **constraint_name**

  约束的名字。约束名长度不超过63个字符。

- **index_name**

  索引名称。

- **storage_parameter**

  表的存储参数的名字。

  分区管理新增的两个选项：

  – PERIOD（interval类型）

    设置分区管理中自动创建分区的周期。

    PERIOD的范围要求以及开启该功能的约束请参考**PERIOD**。

说明

- 在建表时，如果没有设置该参数，可以通过set的方式添加该参数，并开启自动创建分区功能；如果之前已经设置该参数，则通过set的方式修改该参数。
- 用户可以通过reset该参数的方式关闭自动创建分区功能，但是在自动删除分区功能存在的情况下，不支持关闭自动创建分区功能。

– TTL（interval类型）

设置分区管理中自动删除分区的分区过期时间。

TTL的范围要求以及开启该功能的约束请参考**TTL**。

说明

- 在建表时，如果没有设置该参数，可以通过set的方式添加该参数，并开启自动删除分区功能；如果之前已经设置该参数，则通过set的方式修改该参数。
- 用户可以通过reset该参数的方式关闭自动删除分区功能。

- **new_owner**

  表所属新的拥有者的名字。

- **new_tablespace**

  表所属新的表空间名字。

- **column_name，column_1_name， column_2_name**

  现存的或新字段的名称。

- **data_type**

  新字段的类型，或者现存字段的新类型。

- **compress_mode**

  表字段的压缩可选项，当前仅对行存表有效。该子句指定该字段优先使用的压缩算法。

- **collation**

  字段排序规则名称。可选字段COLLATE指定了新字段的排序规则，如果省略，排序规则为新字段的默认类型。

- **USING expression**

  USING子句声明如何从旧的字段值里计算新的字段值；如果省略，缺省从旧类型向新类型的赋值转换。如果从旧数据类型到新类型没有隐含或者赋值的转换，则必须提供一个USING子句。

  说明

  ALTER TYPE的USING选项实际上可以声明涉及该行旧值的任何表达式，即它可以引用除了正在被转换的字段之外其他的字段。这样，就可以用ALTER TYPE语法做非常普遍性的转换。因为这个灵活性，USING表达式并没有作用于该字段的缺省值（如果有的话），结果可能不是缺省表达式要求的常量表达式。这就意味着如果从旧类型到新类型没有隐含或者赋值转换的话，即使存在USING子句，ALTER TYPE也可能无法把缺省值转换成新的类型。在这种情况下，应该用DROP DEFAULT先删除缺省，执行ALTER TYPE，然后使用SET DEFAULT增加一个合适的新缺省值。类似的考虑也适用于涉及该字段的索引和约束。

- **NOT NULL | NULL**

  设置列是否允许空值。

- **integer**

  带符号的整数常值。当使用PERCENT时表示按照表数据的百分比收集统计信息，integer的取值范围为0-100。

- **attribute_option**

  属性选项。

- **PLAIN | EXTERNAL | EXTENDED | MAIN**

  字段存储模式。

  - PLAIN必需用于定长的数值（比如integer）并且是内联的、不压缩的。
  - MAIN用于内联、可压缩的数据。
  - EXTERNAL用于外部保存、不压缩的数据。使用EXTERNAL将令在text和bytea字段上的子字符串操作更快，但付出的代价是增加了存储空间。
  - EXTENDED用于外部的压缩数据，EXTENDED是大多数支持非PLAIN存储的数据的缺省。

- **CHECK (expression)**

  每次将要插入的新行或者将要被更新的行必须使表达式结果为真才能成功，否则会抛出一个异常并且不会修改数据库。

  声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。

  目前，CHECK表达式不能包含子查询也不能引用除当前行字段之外的变量。

- **DEFAULT default_expr**

  给字段指定缺省值。

  缺省表达式的数据类型必须和字段类型匹配。

  缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为NULL 。

  default_expr中若使用后缀操作符（如!），需使用括号括起来。

- **UNIQUE index_parameters**

  UNIQUE (column_name [, ...]) index_parameters

  UNIQUE约束表示表里的一个或多个字段的组合必须在全表范围内唯一。

- **PRIMARY KEY index_parameters**

  PRIMARY KEY (column_name [, ...]) index_parameters

  主键约束表明表中的一个或者一些字段只能包含唯一（不重复）的非NULL值。

- **DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE**

  设置该约束是否可推迟，列存暂不支持。

  - DEFERRABLE：可以推迟到事务结尾使用SET CONSTRAINTS命令检查。
  - NOT DEFERRABLE：在每条命令之后马上检查。
  - INITIALLY IMMEDIATE：那么每条语句之后就立即检查它。
  - INITIALLY DEFERRED：只有在事务结尾才检查它。

- **WITH ({storage_parameter = value} [, ...])**

  为表或索引指定一个可选的存储参数。

- **COMPRESS|NOCOMPRESS**

  - NOCOMPRESS：如果指定关键字NOCOMPRESS则不会修改表的现有压缩特性。
  - COMPRESS：如果指定COMPRESS关键字，则对该表进行批量插入元组时触发该特性。

- **new_table_name**
  修改后新的表名称。
- **new_column_name**
  表中指定列修改后新的列名称。
- **new_constraint_name**
  修改后表约束的新名称。
- **new_schema**
  修改后新的模式名称。
- **CASCADE**
  级联删除依赖于被依赖字段或者约束的对象（比如引用该字段的视图）。
- **RESTRICT**
  如果字段或者约束还有任何依赖的对象，则拒绝删除该字段。这是缺省行为。
- **schema_name**
  表所在的模式名称。

表操作示例

```
DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE CUSTOMER
(
   C_CUSTKEY     BIGINT       ,
   C_NAME        VARCHAR(25)  ,
   C_ADDRESS     VARCHAR(40)  ,
   C_NATIONKEY   INT          ,
   C_PHONE       CHAR(15)     ,
   C_ACCTBAL     DECIMAL(15,2)
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

根据已有唯一索引为表增加主键约束或唯一约束。

先给表CUSTOMER创建唯一索引CUSTOMER_constraint1，然后根据已有唯一索引增加主键约束，并对前面创建的索引rename：

```
CREATE UNIQUE INDEX CUSTOMER_constraint1 ON CUSTOMER(C_CUSTKEY);
ALTER TABLE CUSTOMER ADD CONSTRAINT CUSTOMER_constraint2 PRIMARY KEY USING INDEX
CUSTOMER_constraint1;
```

重命名表约束：

```
ALTER TABLE CUSTOMER RENAME CONSTRAINT CUSTOMER_constraint2 TO CUSTOMER_constraint;
```

删除表约束：

```
ALTER TABLE CUSTOMER DROP CONSTRAINT CUSTOMER_constraint;
```

给表增加一个索引：

```
ALTER TABLE CUSTOMER ADD INDEX CUSTOMER_index(C_CUSTKEY);
```

删除表索引：

```
ALTER TABLE CUSTOMER DROP INDEX CUSTOMER_index;
ALTER TABLE CUSTOMER DROP KEY CUSTOMER_index;
```

给表中某一列添加索引：

```
ALTER TABLE CUSTOMER ADD c_address_id varchar(20) CONSTRAINT ca_address_index CHECK
(c_address_id > 0);
```


给表新增主键约束：

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY(C_CUSTKEY);
```

重命名表：

```
ALTER TABLE CUSTOMER RENAME TO CUSTOMER_t;
```

创建列存表：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH,COLVERSION=2.0)
DISTRIBUTE BY HASH (ca_address_sk);
```

向在一个列存表中添加局部聚簇列：

```
ALTER TABLE customer_address ADD CONSTRAINT customer_address_cluster PARTIAL CLUSTER
KEY(ca_address_sk);
```

删除一个列存表中的局部聚簇列：

```
ALTER TABLE customer_address DROP CONSTRAINT customer_address_cluster;
```

切换列存表的存储格式：

```
ALTER TABLE customer_address SET (COLVERSION = 1.0);
```

修改表的分布方式：

```
ALTER TABLE customer_address DISTRIBUTE BY REPLICATION;
```

修改表模式：

```
CREATE SCHEMA tpcds;
ALTER TABLE customer_address SET SCHEMA tpcds;
```

单表冷热切换：

```
DROP TABLE IF EXISTS cold_hot_table;
CREATE TABLE cold_hot_table(
    W_WAREHOUSE_ID          CHAR(16)            NOT NULL,
    W_WAREHOUSE_NAME        VARCHAR(20)                 ,
    W_STREET_NUMBER         CHAR(10)                    ,
    W_STREET_NAME           VARCHAR(60)                 ,
    W_STREET_ID             CHAR(15)                    ,
    W_SUITE_NUMBER          CHAR(10)                    )
WITH (ORIENTATION = COLUMN, storage_policy = 'LMT:30')
DISTRIBUTE BY HASH (W_WAREHOUSE_ID)
PARTITION BY RANGE(W_STREET_ID)(
    PARTITION P1 VALUES LESS THAN(100000),
    PARTITION P2 VALUES LESS THAN(200000),
    PARTITION P3 VALUES LESS THAN(300000),
    PARTITION P4 VALUES LESS THAN(MAXVALUE)
)ENABLE ROW MOVEMENT;

ALTER TABLE cold_hot_table REFRESH STORAGE;
```

列存分区表修改为冷热表：

```
CREATE table test_1(id int,d_time date)
WITH(ORIENTATION=COLUMN)
DISTRIBUTE BY HASH (id)
```



```
PARTITION BY RANGE (d_time)
(PARTITION p1 START('2022-01-01') END('2022-01-31') EVERY(interval '1 day'));

ALTER TABLE test_1 SET (storage_policy = 'LMT:100');
```

列操作示例

```
DROP TABLE IF EXISTS warehouse_t;
CREATE TABLE warehouse_t
(
   W_WAREHOUSE_SK           INTEGER               NOT NULL,
   W_WAREHOUSE_ID           CHAR(16)              NOT NULL,
   W_WAREHOUSE_NAME         VARCHAR(20)   UNIQUE DEFERRABLE,
   W_WAREHOUSE_SQ_FT        INTEGER                      ,
   W_COUNTY                 VARCHAR(30)                  ,
   W_STATE                  CHAR(2)           DEFAULT 'GA',
   W_ZIP                    CHAR(10)
);
```

向表中增加一个新的字段：

```
ALTER TABLE warehouse_t ADD W_GOODS_CATEGORY int;
```

修改表中列名信息以及列字段信息：

```
ALTER TABLE warehouse_t CHANGE W_GOODS_CATEGORY W_GOODS_CATEGORY2 DECIMAL NOT NULL
COMMENT 'W_GOODS_CATEGORY';
```

给已创建好的表增加主键：

```
ALTER TABLE warehouse_t ADD PRIMARY KEY(w_warehouse_name);
```

重命名列：

```
ALTER TABLE warehouse_t RENAME W_ZIP TO new_W_ZIP;
```

向表中增加多列：

```
ALTER TABLE warehouse_t ADD (W_COMMENT VARCHAR(117) NOT NULL, W_COUNT int);
```

修改表中已存在字段的数据类型，并将字段约束设置为非空：

```
ALTER TABLE warehouse_t MODIFY W_WAREHOUSE_SQ_FT varchar(20) NOT NULL;
```

为表的某列添加not null约束：

```
ALTER TABLE warehouse_t ALTER COLUMN W_COUNTY SET NOT NULL;
```

从表中删除一个字段：

```
ALTER TABLE warehouse_t DROP COLUMN W_STATE;
```

相关链接

CREATE TABLE，**12.101-RENAME TABLE**，**DROP TABLE**

12.21 ALTER TABLE PARTITION

功能描述

修改表分区，包括增删分区、切割分区、合成分区，以及修改分区属性等。

注意事项

- 添加分区的名称不能与该分区表已有分区的名称相同。
- 对于范围分区表，要添加的分区的边界值要和分区表的分区键的类型一致，且要大于分区表的最后一个分区的上边界。
- 对于列表分区表，如果已经定义DEFAULT分区，则不能添加新分区。
- 若文档中未特殊注明，则表明范围分区表和列存分区的语法使用相同。
- 如果目标分区表中已有分区数达到了最大值（32767），则不能继续添加分区。
- 当分区表只有一个分区时，不能删除该分区。
- 删除分区（DROP PARTITION）时会连同分区内数据一起删除。
- 选择分区使用PARTITION FOR()，括号里指定值个数应该与定义分区时使用的列个数相同，并且一一对应。
- Value分区表不支持相应的Alter Partition操作。
- OBS冷热表对于move，exchange，merge，split操作，不支持指定分区表表空间为OBS表空间；执行ALTER语法时，需保持分区数据冷热属性不变（即冷分区操作后为冷分区，热分区操作后为热分区），不支持将冷分区数据切至本地表空间；对于冷分区仅支持默认表空间；merge操作不支持将冷分区与热分区进行合并，exchange操作不支持冷分区交换。

语法格式

- 修改表分区主语法。

```
ALTER TABLE [ IF EXISTS ] { table_name  [*] | ONLY table_name | ONLY ( table_name  )}
  action [, ... ];
```

其中action统指如下分区维护子语法。当存在多个分区维护子句时，保证了分区的连续性，无论这些子句的排序如何，GaussDB(DWS)总会先执行DROP PARTITION再执行ADD PARTITION操作，最后顺序执行其它分区维护操作。

```
  exchange_clause |
  row_clause |
  merge_clause |
  modify_clause |
  split_clause |
  add_clause |
  drop_clause
```

  - exchange_clause子语法用于把普通表的数据迁移到指定的分区。

```
EXCHANGE PARTITION { ( partition_name ) | FOR ( partition_value [, ...] ) }
   WITH TABLE {[ ONLY ] ordinary_table_name | ordinary_table_name * | ONLY
( ordinary_table_name )}
   [ { WITH | WITHOUT } VALIDATION ] [ VERBOSE ]
```

    进行交换的普通表和分区表必须满足如下条件：

    - 普通表和分区表的列数目相同，对应列的信息严格一致，包括：列名、列的数据类型、列约束、列的Collation信息、列的存储参数、列的压缩信息、已删除字段的数据类型等。
    - 普通表和分区表的表压缩信息严格一致。
    - 普通表和分区表的分布列信息严格一致。
    - 普通表和分区表的索引个数相同，且对应索引的信息严格一致。
    - 普通表和分区表的表约束个数相同，且对应表约束的信息严格一致。

- 普通表不可以是临时表和unlogged表。
- 普通表和分区表必须在同一个逻辑集群或节点组（NodeGroup）中。
- 如果行存分区表中最后一个有效字段后的其他字段全部被删除，在不考虑这些删除字段的情况下，分区表与普通表字段信息一致时，分区表和普通表可以进行交换。
- 列存普通表和列存分区表的表级参数colversion必须一致：禁止colversion2.0与colversion1.0执行交换分区操作。

完成交换后，普通表和分区表的数据被置换，同时普通表和分区表的表空间信息被置换。此时，普通表和分区表的统计信息变得不可靠，需要对普通表和分区表重新执行analyze。

- row_clause子语法用于设置分区表的行迁移开关。
  ```
  { ENABLE | DISABLE } ROW MOVEMENT
  ```
- merge_clause子语法用于把多个分区合并成一个分区。
  ```
  MERGE PARTITIONS { partition_name } [, ...] INTO PARTITION partition_name
  ```

须知

- INTO关键字前的分区称为源分区，INTO关键字后的分区称为目标分区。
- 源分区个数不能小于2个，不能大于32个。
- 源分区名称不能重复。
- 源分区不能存在unusable的索引，否则执行会报错。
- 目标分区名只能跟最后一个源分区的名称相同，或者跟表的所有分区名都不相同。
- 目标分区的边界是所有源分区边界的并集。
- 对于范围分区表，所有的源分区必须是边界连续的分区。
- 对于列表分区，如果源分区中包含DEFAULT分区，那么目标分区的边界也是DEFAULT。

- modify_clause子语法用于设置分区索引是否可用。
  ```
  MODIFY PARTITION partition_name { UNUSABLE LOCAL INDEXES | REBUILD UNUSABLE LOCAL INDEXES }
  ```
- split_clause子语法用于把一个分区切割成多个分区。

  范围分区的split_clause语法如下：

  ```
  SPLIT PARTITION { partition_name | FOR ( partition_value [, ...] ) } { split_point_clause | no_split_point_clause }
  ```

  - 指定切割点split_point_clause的语法为：
    ```
    AT ( partition_value ) INTO ( PARTITION partition_name , PARTITION partition_name )
    ```

    须知

    切割点的大小要位于正在被切割的分区的分区键范围内，指定切割点的方式只能把一个分区切割成两个新分区。

- 不指定切割点no_split_point_clause的语法为：
```
INTO { ( partition_less_than_item [, ...] ) | ( partition_start_end_item [, ...] ) }
```

须知

- 不指定切割点的方式，partition_less_than_item指定的第一个新分区的分区键要大于正在被切割的分区的前一个分区（如果存在的话）的分区键，partition_less_than_item指定的最后一个分区的分区键要等于正在被切割的分区的分区键大小。
- 不指定切割点的方式，partition_start_end_item指定的第一个新分区的起始点（如果存在的话）必须等于正在被切割的分区的前一个分区（如果存在的话）的分区键，partition_start_end_item指定的最后一个分区的终止点（如果存在的话）必须等于正在被切割的分区的分区键。
- partition_less_than_item支持的分区键个数最多为4，而partition_start_end_item仅支持1个分区键，其支持的数据类型参见**Partition Key**。
- 在同一语句中partition_less_than_item和partition_start_end_item两者不可同时使用；不同split语句之间没有限制。

- 分区项partition_less_than_item的语法为：
```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE }  [, ...] )
```

- 分区项partition_start_end_item的语法为，其约束参见**START END语法描述**。
```
PARTITION partition_name {
      {START(partition_value) END (partition_value) EVERY (interval_value)} |
      {START(partition_value) END ({partition_value | MAXVALUE})} |
      {START(partition_value)} |
      {END({partition_value | MAXVALUE})}
}
```

列表分区的split_clause语法如下：
```
SPLIT PARTITION { partition_name | FOR ( partition_value [, ...] ) } { split_values_clause | split_no_values_clause }
```

- 指定切割点的split_values_clause的语法为：
```
VALUES ( { (partition_value) [, ...] } | DEFAULT ) INTO ( PARTITION partition_name ，PARTITION partition_name  )
```

须知

- 如果源分区不是**DEFAULT分区**，那么切割点所指定的边界是源分区边界的一个非空真子集；如果源分区是DEFAULT分区，那么切割点所指定的边界不能和其它非DEFAULT分区的边界存在重叠。
- 切割点的指定的边界是INTO关键字后面的第一个分区的边界，源分区边界与切割点的指定的边界的差集是第二个分区的边界。
- 当源分区是DEFAULT分区时，第二个分区的边界还是DEFAULT。

- 不指定切割点的split_no_values_clause的语法为：
```
INTO ( list_partition_item [, ....], PARTITION partition_name )
```

须知

- 此处的**list_partition_item**和创建列表分区表的时候指定分区的语法一样，除了此处的分区定义中边界值不能为DEFAULT。
- 除了最后一个分区，其他分区需要显式定义边界，定义的边界不能是DEFAULT，并且必须是源分区边界的非空真子集。最后一个分区的边界是源分区边界与其它分区边界的差集，且最后一个分区的边界为空(即差集不能为空集)。
- 如果源分区是DEFAULT分区，则最后一个分区的边界为DEFAULT。

– add_clause子语法用于为指定的分区表添加一个或多个分区。

范围分区的add_clause语法如下：

```
ADD { partition_less_than_item... | partition_start_end_item }
```

须知

- 使用**partition_less_than_item**语法时，分区表必须是范围分区表，否则执行会报错。
- 此处**partition_less_than_item**和创建范围分区表的时候指定分区的语法一样。
- 当前分区表的最后一个分区的边界为MAXVALUE，不允许添加新的分区，否则执行会报错。

列表分区的add_clause语法如下：

```
ADD list_partition_item
```

须知

- 使用**list_partition_item**语法时，分区表必须是列表分区表，否则执行会报错
- 此处的**list_partition_item**和创建列表分区表的时候指定分区的语法一样
- 当前分区表存在DEFAULT分区时，不允许添加新的分区动作，否则执行会报错

– drop_clause子语法用于删除分区表中的指定分区。

```
DROP PARTITION { partition_name | FOR ( partition_value [, ...] ) }
```

– drop_clause子语法支持删除多个分区语法。（8.1.3.100及以上集群版本支持。）

```
DROP PARTITION { partition_name [, ... ] }
```

- 修改表分区名的语法。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name )}
    RENAME PARTITION { partition_name | FOR ( partition_value [, ...] ) } TO partition_new_name;
```

参数说明

- **table_name**

  分区表名。

取值范围：已存在的分区表名。

- **partition_name**

  分区名。

  取值范围：已存在的分区名。

- **partition_value**

  分区键值。

  通过PARTITION FOR (partition_value [, ...])子句指定的这一组值，可以唯一确定一个分区。

  取值范围：需要进行重命名的分区的分区键的取值范围。

- **UNUSABLE LOCAL INDEXES**

  设置该分区上的所有索引不可用。

- **REBUILD UNUSABLE LOCAL INDEXES**

  重建该分区上的所有索引。

- **ENABLE/DISABLE ROW MOVEMENT**

  行迁移开关。

  取值范围：

  - ENABLE：打开行迁移开关。
  - DISABLE：关闭行迁移开关。

  默认是关闭状态。

  说明

  - ENABLE ROW MOVEMENT开启则允许跨分区更新，但此时如果有SELECT FOR UPDATE查询该分区表并发执行，存在查询结果瞬时不一致的可能性，需要谨慎使用。
  - 如果进行UPDATE操作时，更新了元组在分区键上的值，造成了该元组所在分区发生变化，就会根据该开关给出报错信息，或者进行元组在分区间的转移。

- **ordinary_table_name**

  进行迁移的普通表的名称。

  取值范围：已存在的普通表名。

- **{ WITH | WITHOUT } VALIDATION**

  在进行数据迁移时，是否检查普通表中的数据满足指定分区的分区键范围。

  取值范围：

  - WITH：对于普通表中的数据要检查是否满足分区的分区键范围，如果有数据不满足，则报错。
  - WITHOUT：对于普通表中的数据不检查是否满足分区的分区键范围。

  默认是WITH状态。

  由于检查比较耗时，特别是当数据量很大的情况下更甚。所以在保证当前普通表中的数据满足分区的分区键范围时，可以加上WITHOUT来指明不进行检查。

- **VERBOSE**

  在VALIDATION是WITH状态时，如果检查出普通表有不满足要交换分区的分区键范围的数据，那么把这些数据插入到正确的分区，如果路由不到任何分区，再报错。

须知

只有在VALIDATION是WITH状态时，才可以指定VERBOSE。

- **partition_new_name**

  分区的新名字。

  取值范围：字符串，要符合标识符的命名规范。

示例

创建范围分区表customer_address：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(100),
        PARTITION P2 VALUES LESS THAN(200),
        PARTITION P3 VALUES LESS THAN(300)
);
```

创建list分区表：

```
DROP TABLE IF EXISTS data_list;
CREATE TABLE data_list(
    id int,
    time int,
     sarlay decimal(12,2)
)PARTITION BY LIST (time)(
        PARTITION P1 VALUES (202209),
        PARTITION P2 VALUES (202210,202208),
        PARTITION P3 VALUES (202211),
        PARTITION P4 VALUES (202212),
        PARTITION P5 VALUES (202301)
);
```

- modify_clause子句用于设置分区索引是否可用。

  给分区表customer_address创建LOCAL索引student_grade_index，并指定分区的索引名称：

  ```
  CREATE INDEX customer_address_index ON customer_address(ca_address_id) LOCAL
  (
        PARTITION P1_index,
        PARTITION P2_index,
        PARTITION P3_inde
  );
  ```

  重建分区表customer_address中分区P1上的所有索引：

  ```
  ALTER TABLE customer_address MODIFY PARTITION P1 REBUILD UNUSABLE LOCAL INDEXES;
  ```

  设置分区表customer_address的分区P3上的所有索引不可用：

  ```
  ALTER TABLE customer_address MODIFY PARTITION P3 UNUSABLE LOCAL INDEXES;
  ```

- add_clause子句用于为指定的分区表添加一个或多个分区。

  为范围分区表customer_address增加分区：

```
ALTER TABLE customer_address ADD PARTITION P5 VALUES LESS THAN (500);
```

为范围分区表customer_address增加分区: [500, 600), [600, 700)：

```
ALTER TABLE customer_address ADD PARTITION p6 START(500) END(700) EVERY(100);
```

为范围分区表customer_address增加MAXVALUE分区p7：

```
ALTER TABLE customer_address ADD PARTITION p7 END(MAXVALUE);
```

为列表分区表增加分区P6：

```
ALTER TABLE data_list ADD PARTITION P6 VALUES (202302,202303);
```

- split_clause子句用于把一个分区切割成多个分区。

  将范围分区表customer_address的P7分区以800为分割点切分：

  ```
  ALTER TABLE customer_address SPLIT PARTITION P7 AT(800) INTO (PARTITION P6a,PARTITION P6b);
  ```

  将范围分区表customer_address中400所在的分区分割成多个分区：

  ```
  ALTER TABLE customer_address SPLIT PARTITION FOR(400) INTO(PARTITION p_part START(300)
  END(500) EVERY(100));
  ```

  将列表分区表data_list的分区P2分割成p2a和p2b两个分区：

  ```
  ALTER TABLE data_list SPLIT PARTITION P2 VALUES(202210) INTO (PARTITION p2a,PARTITION p2b);
  ```

- exchange_clause子句：把普通表的数据迁移到指定的分区。

  下面示例演示了把一个普通表math_grade数据迁移到分区表student_grade 中分区（math）的操作。创建分区表student_grade ：

  ```
  CREATE TABLE student_grade (
        stu_name     char(5),
        stu_no       integer,
        grade        integer,
        subject      varchar(30)
  )
  PARTITION BY LIST(subject)
  (
        PARTITION gym VALUES('gymnastics'),
        PARTITION phys VALUES('physics'),
        PARTITION history VALUES('history'),
        PARTITION math VALUES('math')
  );
  ```

  添加数据到分区表student_grade中：

  ```
  INSERT INTO student_grade VALUES
        ('Ann', 20220101, 75, 'gymnastics'),
        ('Jeck', 20220103, 60, 'math'),
        ('Anna', 20220108, 56, 'history'),
        ('Jann', 20220107, 82, 'physics'),
        ('Molly', 20220104, 91, 'physics'),
        ('Sam', 20220105, 72, 'math');
  ```

  查询分区表student_grade的math分区记录：

  ```
  SELECT * FROM student_grade PARTITION (math);
   stu_name |  stu_no  | grade | subject
  ----------+----------+-------+---------
   Jeck     | 20220103 |    60 | math
   Sam      | 20220105 |    72 | math
  (2 rows)
  ```

  创建一个与分区表student_grade定义匹配的普通表math_grade：

  ```
  CREATE TABLE math_grade
  (
        stu_name     char(5),
        stu_no       integer,
        grade        integer,
        subject      varchar(30)
  );
  ```

  添加了数据到表math_grade中。数据与分区表student_grade的math分区分区规则一致：


```
INSERT INTO math_grade VALUES
      ('Ann', 20220101, 75, 'math'),
      ('Jeck', 20220103, 60, 'math'),
      ('Anna', 20220108, 56, 'math'),
      ('Jann', 20220107, 82, 'math');
```

将普通表math_grade数据迁移到分区表student_grade 中分区（math）：

```
ALTER TABLE student_grade EXCHANGE PARTITION (math) WITH TABLE math_grade;
```

对分区表student_grade的查询表明表math_grade中的数据已和分区math中的数据交换：

```
SELECT * FROM student_grade PARTITION (math);
 stu_name |  stu_no  | grade | subject
----------+----------+-------+---------
 Anna     | 20220108 |    56 | math
 Jeck     | 20220103 |    60 | math
 Ann      | 20220101 |    75 | math
 Jann     | 20220107 |    82 | math
(4 rows)
```

对表math_grade的查询显示了之前存储在分区math中的记录已被移动到表student_grade中：

```
SELECT * FROM math_grade;
 stu_name |  stu_no  | grade | subject
----------+----------+-------+---------
 Jeck     | 20220103 |    60 | math
 Sam      | 20220105 |    72 | math
(2 rows)
```

- row_clause子句用于设置分区表的行迁移开关。

  打开分区表customer_address的迁移开关：

  ```
  ALTER TABLE customer_address ENABLE ROW MOVEMENT;
  ```

- merge_clause子句用于把多个分区合并成一个分区。

  将范围分区表customer_address的P2，P3两个分区合并为一个分区：

  ```
  ALTER TABLE customer_address MERGE PARTITIONS P2, P3 INTO PARTITION P_M;
  ```

- drop_clause子句用于删除分区表中的指定分区。

  删除分区表customer_address的分区P2：

  ```
  ALTER TABLE customer_address DROP PARTITION P2;
  ```

  删除分区表customer_address的多个分区P6a, P6b：

  ```
  ALTER TABLE customer_address DROP PARTITION P6a, P6b;
  ```

相关链接

CREATE TABLE PARTITION，**DROP TABLE**

12.22 ALTER TEXT SEARCH CONFIGURATION

功能描述

更改文本搜索配置的定义。用户可以将映射从字串类型调整为字典，或者改变配置的名称或者所有者，或者修改搜索配置的配置参数。

注意事项

- 当一个搜索配置已经被引用（如被用来创建索引），则不允许用户修改此文本搜索配置。

- 要使用ALTER TEXT SEARCH CONFIGURATION，用户必须是配置的所有者。

语法格式

- 增加文本搜索配置字串类型映射。

```
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ];
```

- 修改文本搜索配置字典。

```
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary;
```

- 修改文本搜索配置字串类型。

```
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ];
```

- 更改文本搜索配置字典。

```
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary;
```

- 删除文本搜索配置字串类型映射。

```
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ];
```

- 重命名文本搜索配置所有者。

```
ALTER TEXT SEARCH CONFIGURATION name OWNER TO new_owner;
```

- 重命名文本搜索配置名称。

```
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name;
```

- 重命名文本搜索配置命名空间。

```
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema;
```

- 修改文本搜索配置属性。

```
ALTER TEXT SEARCH CONFIGURATION name SET ( { configuration_option = value } [, ...] );
```

- 重置文本搜索配置属性。

```
ALTER TEXT SEARCH CONFIGURATION name RESET ( {configuration_option} [, ...] );
```

说明

- ADD MAPPING FOR选项为文本搜索配置增加字串类型映射；如果ADD MAPPING FOR后面任何一个字串类型的映射已经存在于此文本搜索配置中，那么系统将会报错。
- ALTER MAPPING FOR选项会首先清除已有的字串类型映射，然后添加指定的字串类型映射。
- ALTER MAPPING REPLACE ... WITH ... 与ALTER MAPPING FOR ... REPLACE ... WITH ...选项会直接使用new_dictionary替换old_dictionary。需要注意的是，只有pg_ts_config_map系统表中存在maptokentype与old_dictionary对应关系的元组时，才能更新成功，否则不会成功，也不会有任何提示信息返回。
- DROP MAPPING FOR选项会删除当前文本搜索配置中指定的字串类型映射。 如果没有指定IF EXISTS选项，当DROP MAPPING FOR选项指定的字串类型映射在文本搜索配置中不存在时，数据库会报错。

参数说明

- **name**

  已有文本搜索配置的名称（可以有模式修饰）。

- **token_type**

  与配置的语法解析器关联的字串类型的名称。详细信息参见**文本搜索解析器**。

- **dictionary_name**
  文本搜索字典名称。 如果有多个字典，则它们会按指定的顺序搜索。
- **old_dictionary**
  映身中拟被替换的文本搜索字典名称。
- **new_dictionary**
  替换old_dictionary的文本搜索字典的名称。
- **new_owner**
  文本搜索配置的新所有者。
- **new_name**
  文本搜索配置的新名称。
- **new_schema**
  文本搜索配置的新模式名。
- **configuration_option**
  文本搜索配置项。详细信息参见**CREATE TEXT SEARCH CONFIGURATION**。
- **value**
  文本搜索配置项的值。

示例

创建文本搜索配置：

```
DROP TEXT SEARCH CONFIGURATION IF EXISTS ngram1;
CREATE TEXT SEARCH CONFIGURATION ngram1 (parser=ngram) WITH (gram_size = 2, grapsymbol_ignore = false);
```

给文本搜索类型ngram1添加类型映射：

```
ALTER TEXT SEARCH CONFIGURATION ngram1 ADD MAPPING FOR multisymbol WITH simple;
```

修改文本搜索配置的所有者：

```
CREATE ROLE joe password '{Password}';
ALTER TEXT SEARCH CONFIGURATION ngram1 OWNER TO joe;
```

修改文本搜索配置的schema：

```
ALTER TEXT SEARCH CONFIGURATION ngram1 SET SCHEMA joe;
```

重命名文本搜索配置：

```
ALTER TEXT SEARCH CONFIGURATION joe.ngram1 RENAME TO ngram_1;
```

删除类型映射：

```
ALTER TEXT SEARCH CONFIGURATION joe.ngram_1 DROP MAPPING IF EXISTS FOR multisymbol;
```

创建文本搜索配置：

```
DROP TEXT SEARCH CONFIGURATION IF EXISTS english_1;
CREATE TEXT SEARCH CONFIGURATION english_1 (parser=default);
```

增加文本搜索配置字串类型映射语法：

```
ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR word WITH simple,english_stem;
```

增加文本搜索配置字串类型映射语法：

```
ALTER TEXT SEARCH CONFIGURATION english_1 ADD MAPPING FOR email WITH english_stem,
french_stem;
```

修改文本搜索配置字串类型映射语法：

```
ALTER TEXT SEARCH CONFIGURATION english_1 ALTER MAPPING REPLACE french_stem with german_stem;
```

查询文本搜索配置相关信息：

```
SELECT b.cfgname,a.maptokentype,a.mapseqno,a.mapdict,c.dictname FROM pg_ts_config_map
a,pg_ts_config b, pg_ts_dict c WHERE a.mapcfg=b.oid AND a.mapdict=c.oid AND b.cfgname='english_1'
ORDER BY 1,2,3,4,5;
  cfgname  | maptokentype | mapseqno | mapdict |    dictname
-----------+--------------+----------+---------+--------------
 english_1 |            2 |        1 |    3765 | simple
 english_1 |            2 |        2 |   12960 | english_stem
 english_1 |            4 |        1 |   12960 | english_stem
 english_1 |            4 |        2 |   12966 | german_stem
(4 rows)
```

相关链接

CREATE TEXT SEARCH CONFIGURATION， **DROP TEXT SEARCH CONFIGURATION**

12.23 ALTER TEXT SEARCH DICTIONARY

功能描述

修改全文检索词典的相关定义，包括参数、名称、所有者以及模式等。

注意事项

- 预定义词典不支持ALTER操作。
- 只有词典的所有者可以执行ALTER操作，系统管理员默认拥有此权限。
- 创建或修改词典之后，任何对于filepath路径下用户自定义的词典定义文件的修改，将不会影响到数据库中的词典。如果需要在数据库中使用这些修改，需使用ALTER TEXT SEARCH DICTIONARY语句更新对应词典的定义文件。

语法格式

- 修改词典定义。
  ```
  ALTER TEXT SEARCH DICTIONARY name (
      option [ = value ] [, ... ]
  );
  ```
- 重命名词典。
  ```
  ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name;
  ```
- 设置词典的所属模式。
  ```
  ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema;
  ```
- 修改词典的所属者。
  ```
  ALTER TEXT SEARCH DICTIONARY name OWNER TO new_owner;
  ```

参数说明

- **name**

  已存在的词典名（可指定模式名，否则默认在当前模式下）。

取值范围：已存在的词典名。

- **option**

  要修改的参数名。与template对应，不同的词典类型具有不同的参数列表，且与指定顺序无关。详细参数说明请见**option**。

  说明

  - 不支持修改词典的TEMPLATE参数值。
  - 不支持仅修改FILEPATH参数而不修改对应的词典定义文件参数。
  - 词典定义文件的文件名仅支持小写字母、数据、下划线混合。

- **value**

  要修改的参数值。如果省略等号（ = ）和value，则表示删除该option的先前设置，使用默认值。

  取值范围：对应option定义。

- **new_name**

  词典的新名称。

  取值范围：符合标识符命名规范的字符串，且最大长度不超过63个字符。

- **new_owner**

  词典新的所有者。

  取值范围：已存在的用户。

- **new_schema**

  词典的新模式。

  取值范围：已存在的模式。

示例

```
CREATE TEXT SEARCH DICTIONARY my_dict (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

更改Snowball类型字典的停用词定义，其他参数保持不变。

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian, FilePath = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg' );
```

更改Snowball类型字典的Language参数，并删除停用词定义。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( Language = dutch， StopWords );
```

更新词典定义，不实际更改任何内容。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

相关链接

CREATE TEXT SEARCH DICTIONARY，DROP TEXT SEARCH DICTIONARY

12.24 ALTER TRIGGER

功能描述

修改触发器定义。

注意事项

只有触发器所在表的所有者可以执行ALTER TRIGGER操作，系统管理员默认拥有此权限。

语法格式

```
ALTER TRIGGER trigger_name ON table_name RENAME TO new_name;
```

参数说明

- **trigger_name**

  要修改的触发器名字。

  取值范围：已存在的触发器。

- **table_name**

  要修改的触发器所在的表名称。

  取值范围：已存在的含触发器的表。

- **new_name**

  修改后的新的触发器名字。

  取值范围：符合标识符命名规范的字符串，最大长度不超过63个字符，且不能与所在表上其他触发器同名。

示例

创建源表及触发表：

```
DROP TABLE IF EXISTS test_trigger_src_tbl;
DROP TABLE IF EXISTS test_trigger_des_tbl;

CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);
CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);
```

创建触发器函数tri_insert_func()：

```
DROP FUNCTION IF EXISTS tri_insert_func;
CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
        $$
        DECLARE
        BEGIN
                INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2, NEW.id3);
                RETURN NEW;
        END
        $$ LANGUAGE PLPGSQL;
```

创建INSERT触发器：

```
CREATE TRIGGER insert_trigger
        BEFORE INSERT ON test_trigger_src_tbl
        FOR EACH ROW
        EXECUTE PROCEDURE tri_insert_func();
```

修改触发器delete_trigger：

```
ALTER TRIGGER insert_trigger ON test_trigger_src_tbl RENAME TO insert_trigger_renamed;
```

禁用触发器insert_trigger：

```
ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER insert_trigger_renamed;
```

禁用当前表test_trigger_src_tbl所有触发器：

```
ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER ALL;
```

相关链接

CREATE TRIGGER，**DROP TRIGGER**，**ALTER TABLE**

12.25 ALTER TYPE

功能描述

修改一个类型的定义。

语法格式

- 修改类型

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE |
RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER }
neighbor_enum_value ]
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value

where action is one of:
    ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE |
RESTRICT ]
```

- 给复合类型增加新的属性。

```
ALTER TYPE name ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE |
RESTRICT ]
```

- 从复合类型删除一个属性。

```
ALTER TYPE name DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
```

- 改变一种复合类型中某个属性的类型。

```
ALTER TYPE name ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE
collation ] [ CASCADE | RESTRICT ]
```

- 改变类型的所有者。

```
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

- 改变类型的名称或是一个复合类型中的一个属性的名称。

```
ALTER TYPE name RENAME TO new_name
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE |
RESTRICT ]
```

- 将类型移至一个新的模式中。
  ```
  ALTER TYPE name SET SCHEMA new_schema
  ```
- 为枚举类型增加一个新值。
  ```
  ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER } neighbor_enum_value ]
  ```
- 重命名枚举类型的一个标签值。
  ```
  ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
  ```

参数说明

- **name**

  一个需要修改的现有的类型的名字(可以有模式修饰) 。
- **new_name**

  该类型的新名称。
- **new_owner**

  新所有者的用户名 。
- **new_schema**

  该类型的新模式 。
- **attribute_name**

  拟增加、更改或删除的属性的名称。
- **new_attribute_name**

  拟改名的属性的新名称。
- **data_type**

  拟新增属性的数据类型，或是拟更改的属性的新类型名。
- **new_enum_value**

  枚举类型新增加的标签值，是一个非空的长度不超过64个字节的字符串。
- **neighbor_enum_value**

  一个已有枚举标签值，新值应该被增加在紧接着该枚举值之前或者之后的位置上。
- **existing_enum_value**

  现有的要重命名的枚举值，是一个非空的长度不超过64个字节的字符串
- **CASCADE**

  自动级联更新需更新类型以及相关联的记录和继承它们的子表。
- **RESTRICT**

  如果需联动更新类型是已更新类型的关联记录，则拒绝更新。这是缺省选项。

须知

- ADD ATTRIBUTE、DROP ATTRIBUTE和ALTER ATTRIBUTE选项可以组合成一个列表同时执行。 例如，在一条命令中同时增加多个属性或是更改多个属性的类型。
- 要使用ALTER TYPE，必须是该类型的所有者。 要修改类型的模式，还必须在新模式上拥有CREATE权限。 要修改所有者，必须是新的所有角色的直接或间接成员， 并且该角色必须在此类型的模式上有CREATE权限。（这些限制强制要求修改所有者不能执行任何通过删除和重建该类型无法实现的操作。 不过，系统管理员拥有以任何方式修改任意类型的所有权。）要增加属性或是修改属性的类型，也必须拥有该类型的USAGE权限。

示例

创建示例复合类型test，枚举类型testdata和用户user_t。

```
CREATE TYPE test AS (col1 int, col text);
CREATE TYPE testdata AS ENUM ('create', 'modify', 'closed');
CREATE USER user_t PASSWORD '{Password}';
```

重命名数据类型：

```
ALTER TYPE test RENAME TO test1;
```

修改用户定义类型test1的所有者为user_t：

```
ALTER TYPE test1 OWNER TO user_t;
```

把用户定义类型test1的模式改为user_t：

```
ALTER TYPE test1 SET SCHEMA user_t;
```

给数据类型test1增加一个新的属性f3：

```
ALTER TYPE user_t.test1 ADD ATTRIBUTE col3 int;
```

给枚举类型testdata添加一个标签值：

```
ALTER TYPE testdata ADD VALUE IF NOT EXISTS 'regress' BEFORE 'closed';
```

重命名枚举类型testdata的一个标签值：

```
ALTER TYPE testdata RENAME VALUE 'create' TO 'new';
```

相关链接

CREATE TYPE，**DROP TYPE**

12.26 ALTER USER

功能描述

修改数据库用户的属性。

注意事项

ALTER USER修改的会话参数只针对指定的用户，且在下一次会话中有效。

语法格式

- 修改用户的权限等信息。

```
ALTER USER user_name [ [ WITH ] option [ ... ] ];
```

其中option子句为。

```
{ CREATEDB | NOCREATEDB }
    | { CREATEROLE | NOCREATEROLE }
    | { INHERIT | NOINHERIT }
    | { AUDITADMIN | NOAUDITADMIN }
    | { SYSADMIN | NOSYSADMIN }
    | { USEFT | NOUSEFT }
    | { LOGIN | NOLOGIN }
    | { REPLICATION | NOREPLICATION }
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD { 'password' | DISABLE }
    | [ ENCRYPTED | UNENCRYPTED ] IDENTIFIED BY { 'password' [ REPLACE 'old_password' ] |
DISABLE }
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_cluster_name
    | ACCOUNT { LOCK | UNLOCK }
    | PGUSER
    | AUTHINFO 'authinfo'
    | PASSWORD EXPIRATION period
```

- 修改用户名。

```
ALTER USER user_name
    RENAME TO new_name;
```

- 修改与用户关联的指定会话参数值。

```
ALTER USER user_name [ IN DATABASE database_name ]
    SET configuration_parameter { { TO | = } { value | DEFAULT } | FROM CURRENT };
```

- 重置与用户关联的指定会话参数值。

```
ALTER USER user_name [ IN DATABASE database_name ]
    RESET { configuration_parameter | ALL };
```

参数说明

- **user_name**

  现有用户名。

  取值范围：已存在的用户名。

- **new_name**

  用户的新名称。

  取值范围：字符串，要符合标识符的命名规范，且最多为63个字符。

- **CREATEDB | NOCREATEDB**

  决定一个新用户是否能创建数据库。

  新用户没有创建数据库的权限。缺省为NOCREATEDB。

- **CREATEROLE | NOCREATEROLE**

  决定一个用户是否可以创建新用户或角色（也就是执行CREATE ROLE和CREATE USER）。一个拥有CREATEROLE权限的用户也可以修改和删除其他用户。

  缺省为NOCREATEROLE。

- **INHERIT | NOINHERIT**

  决定一个用户是否“继承”它所在组的用户的权限。不推荐使用。

- **AUDITADMIN | NOAUDITADMIN**

  定义用户是否有审计管理属性。

  缺省为NOAUDITADMIN。

- **SYSADMIN | NOSYSADMIN**

  决定一个新用户是否为“系统管理员”，具有SYSADMIN属性的用户拥有系统最高权限。

  缺省为NOSYSADMIN。

- **USEFT | NOUSEFT**

  决定一个新用户是否能操作外表，包括：新建外表、删除外表、修改外表、读写外表。

  新用户没有操作外表的权限。

  缺省为NOUSEFT，表示新建用户默认不能操作外表。

- **LOGIN | NOLOGIN**

  具有LOGIN属性的用户才可以登录数据库。

  缺省为NOLOGIN。

- **REPLICATION | NOREPLICATION**

  定义用户是否允许流复制或设置系统为备份模式。REPLICATION属性是特定的用户，仅用于复制。

  缺省为NOREPLICATION。

- **INDEPENDENT | NOINDEPENDENT**

  定义私有、独立的用户。具有INDEPENDENT属性的用户，管理员对其进行的控制、访问的权限被分离，具体规则如下：

  - 未经INDEPENDENT用户授权，管理员无权对其表对象进行增、删、查、改、复制、授权操作。
  - 未经INDEPENDENT用户授权，管理员无权修改INDEPENDENT用户的继承关系。
  - 管理员无权修改INDEPENDENT用户的表对象的属主。
  - 管理员无权去除INDEPENDENT用户的INDEPENDENT属性。
  - 管理员无权修改INDEPENDENT用户的数据库密码，INDEPENDENT用户需管理好自身密码，密码丢失无法重置。
  - 管理员属性用户不允许定义修改为INDEPENDENT属性。

- **VCADMIN | NOVCADMIN**

  定义逻辑集群管理员用户。具有逻辑集群管理员属性的用户，和普通用户相比，有如下额外权限：

  - 在所关联逻辑集群中创建、修改和删除资源池的权限。
  - 将所关联的逻辑集群的访问权限授予其他用户或角色，或回收其他用户或角色对关联逻辑集群的访问权限。

- **CONNECTION LIMIT**

  声明该用户在单个CN上可以使用的并发连接数量。

  取值范围：整数，>=-1，缺省值为-1，表示没有限制。

须知

为保证集群正常使用，connection limit的最小值是集群中CN的数目。在集群做ANALYZE时，其他CN节点会连接当前做ANALYZE的CN节点来同步元数据。例如集群中有3个CN节点，那么connection limit应该设置为>=3。

- **ENCRYPTED | UNENCRYPTED**

  控制密码存储在系统表里的密码是否加密。（如果没有指定，那么缺省的行为由配置参数password_encryption_type控制。）按照产品安全要求，密码必须加密存储，所以，UNENCRYPTED在GaussDB(DWS)中禁止使用。因为系统无法对指定的加密密码字符串进行解密，所以如果目前的密码字符串已经是用SHA256加密的格式，则会继续照此存放，而不管是否声明了ENCRYPTED或UNENCRYPTED。这样就允许在dump/restore的时候重新加载加密的密码。

  – password

    登录密码。

    密码规则：密码默认不少于8个字符；不能与用户名及用户名倒序相同；至少包含大写字母（A-Z）、小写字母（a-z）、数字（0-9）、非字母数字字符（~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。使用范围外的字符会收到告警，但依然允许创建。

    取值范围：字符串。

  – DISABLE

    默认情况下，用户可以更改自己的密码，除非密码被禁用。要禁用用户的密码，请指定DISABLE。禁用某个用户的密码后，将从系统中删除该密码，此类用户只能通过外部认证来连接数据库，例如：IAM认证、kerberos或ldap认证。只有管理员才能启用或禁用密码。普通用户不能禁用初始用户的密码。要启用密码，请运行ALTER USER并指定密码。

- **VALID BEGIN**

  设置用户生效的时间戳。如果省略了该子句，用户无有效开始时间限制。

- **VALID UNTIL**

  设置用户失效的时间戳。如果省略了该子句，用户无有效结束时间限制。

- **RESOURCE POOL**

  设置用户使用的resource pool名字，该名字属于系统表：pg_resource_pool。

- **USER GROUP 'groupuser'**

  创建一个user的子用户。

- **PERM SPACE**

  设置用户永久表存储空间限额。

  space_limit：永久表存储空间上限。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **TEMP SPACE**

  设置用户临时表存储空间限额。

  tmpspacelimit：临时表存储空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **SPILL SPACE**

  设置用户算子落盘空间限额。

spillspacelimit：算子落盘空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **NODE GROUP**

  设置用户关联的逻辑集群名称。如果需要关联的逻辑集群名称包含大写字符或特殊字符，指定逻辑集群名称时需要加双引号。

- **ACCOUNT LOCK | ACCOUNT UNLOCK**
  - ACCOUNT LOCK：锁定账户，禁止登录数据库。
  - ACCOUNT UNLOCK：解锁账户，允许登录数据库。

- **PGUSER**

  该属性用于兼容开源Postgres的连接通讯，开源的Postgres客户端接口（推荐使用Postgres 9.2.19版本的相关客户端接口）可以使用具有该属性的数据库用户连接数据库。

  当前版本不允许修改用户的PGUSER属性。

须知

该属性只用于兼容连接过程，而由于本产品与Postgres的内核差异导致的不兼容，不在此属性控制范围内。

由于具有PGUSER属性的用户的认证方式与其他用户不同，开源客户端的报错信息可能导致数据库用户PGUSER属性被枚举，建议使用本产品自有的客户端。例如：

```
#normaluser是不具有PGUSER属性的用户，psql是Postgres的客户端工具
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser
psql: authentication method 10 not supported

#pguser用户是具有PGUSER属性的用户
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser
Password for user pguser:
```

- **AUTHINFO 'authinfo'**

  该属性用于指定用户认证类型，authinfo为类型说明字符串，大小写敏感。当前仅支持LDAP类型，对应的类型说明字符串为'ldap'。LDAP属于外部认证，故需要同时指定PASSWORD DISABLE。

须知

- authinfo可以加上LDAP认证的额外信息，比如LDAP认证中的fulluser，fulluser等同于ldapprefix+username+ldapsuffix。当authinfo为'ldap'，表示用户认证类型为LDAP，此时ldapprefix和ldapsuffix的信息由pg_hba.conf中匹配的记录提供。
- ALTER ROLE时不允许用户切换认证类型，仅允许LDAP用户修改LDAP属性。

- **PASSWORD EXPIRATION period**

  声明该用户的登录密码过期天数，登录密码过期之前用户需要及时修改密码。登录密码过期后用户无法登录，需要请管理员设置新的登录密码后登录。

  取值范围：整数，-1~999。缺省值为-1，表示没有过期限制；0表示用户登录密码立即过期。

示例

创建示例用户u1：

```
DROP USER IF EXISTS u1;
CREATE USER u1 PASSWORD '{Password}';
```

修改用户u1的登录密码：

```
ALTER USER u1 IDENTIFIED BY '{new_Password}' REPLACE '{Password}';
```

为用户u1追加CREATEROLE权限：

```
ALTER USER u1 CREATEROLE;
```

将与用户u1关联的会话参数enable_seqscan的值设置为on， 设置成功后，在下一会话中生效：

```
ALTER USER u1 SET enable_seqscan TO on;
```

重置u1的enable_seqscan参数：

```
ALTER USER u1 RESET enable_seqscan;
```

锁定u1账户：

```
ALTER USER u1 ACCOUNT LOCK;
```

相关链接

CREATE ROLE，**CREATE USER**，**DROP USER**

12.27 ALTER VIEW

功能描述

ALTER VIEW更改视图的各种辅助属性。（如果用户想要更改视图的查询定义，要使用CREATE OR REPLACE VIEW。）

注意事项

- 用户必须是视图的所有者才可以使用ALTER VIEW。
- 要改变视图的模式，用户必须要有新模式的CREATE权限。
- 要改变视图的所有者，用户必须是新所属角色的直接或者间接的成员，并且此角色必须有视图模式的CREATE权限。
- 管理员用户可以更改任何视图的所属关系。

语法格式

- 设置视图列的默认值。
  ```
  ALTER VIEW [ IF EXISTS ] view_name
      ALTER [ COLUMN ] column_name SET DEFAULT expression;
  ```
- 取消列视图列的默认值。
  ```
  ALTER VIEW [ IF EXISTS ] view_name
      ALTER [ COLUMN ] column_name DROP DEFAULT;
  ```
- 修改视图的所有者。
  ```
  ALTER VIEW [ IF EXISTS ] view_name
      OWNER TO new_owner;
  ```

- 重命名视图。
```
ALTER VIEW [ IF EXISTS ] view_name
    RENAME TO new_name;
```
- 设置视图的所属模式。
```
ALTER VIEW [ IF EXISTS ] view_name
    SET SCHEMA new_schema;
```
- 设置视图的选项。
```
ALTER VIEW [ IF EXISTS ] view_name
    SET ( { view_option_name [ = view_option_value ] } [, ... ] );
```
- 重置视图的选项。
```
ALTER VIEW [ IF EXISTS ] view_name
    RESET ( view_option_name [, ... ] );
```
- 重建本视图及上层和下层依赖的无效视图。
```
ALTER VIEW [ IF EXISTS ] view_name
    REBUILD;
```
- 重建本视图及下层依赖的视图。
```
ALTER VIEW [ IF EXISTS ] ONLY view_name
    REBUILD;
```

参数说明

- **IF EXISTS**

  使用这个选项，如果视图不存在时不会产生错误，仅有会有一个提示信息。
- **view_name**

  视图名称，可以用模式修饰。

  取值范围：字符串，符合标识符命名规范。
- **column_name**

  可选的名字列表，视图的字段名。如果没有给出，字段名取自查询中的字段名。

  取值范围：字符串，符合标识符命名规范。
- **SET/DROP DEFAULT**

  设置或删除一个列的缺省值，该参数暂无实际意义。
- **new_owner**

  视图新所有者的用户名称。
- **new_name**

  视图的新名称。
- **new_schema**

  视图的新模式。
- **view_option_name [= view_option_value]**

  该子句为视图指定一个可选的参数。

  目前view_option_name支持的参数仅有security_barrier，当VIEW试图提供行级安全时，应使用该参数。

  取值范围：boolean类型，TRUE、FALSE。
- **REBUILD**

  该子句用于视图解耦，可使用已保存的原始语句重新创建视图，恢复依赖关系。REBUILD注意事项如下：
  - 重建视图会从当前视图开始，依次向后级联刷新与其关联的所有视图，如果其依赖的前向视图也为不可用状态，会触发自动重建。

  - 不支持对有依赖关系的临时表及临时视图的解耦DROP，可以对没有依赖关系的临时视图进行REBUILD操作。
  - 支持视图模式名称及视图名称的修改，REBUILD按照最新的名称重建，但是query部分保留原始定义。
  - 基表字段类型仅支持大类（字符型、数字型、时间类型等）范围内的修改；当基表添加字段时，视图不会置为无效，且定义不变。
  - 无效视图备份时以注释形式导出，恢复时需要自行手动处理。
  - GUC参数view_independent设置为on时，支持视图自动重建。

  说明

  以下场景会触发上层级联视图无效：

  - DROP TABLE/VIEW
  - RENAME TABLE/VIEW
  - ALTER TABLE DROP COLUMN
  - ALTER TABLE CHANGE/ALTER COLUMN TYPE
  - ALTER TABLE CHANGE/ALTER COLUMN NAME
  - ALTER TABLE/VIEW NAMESPACE
  - ALTER TABLE/VIEW RENAME

- **ONLY**

  控制视图重建的范围，只重建视图及其所依赖的视图。当GUC参数view_independent设置为on时，此功能可正常使用。

示例

创建示例视图myview：

```
CREATE OR REPLACE VIEW myview AS
    SELECT * FROM pg_tablespace WHERE spcname = 'pg_default';
```

修改视图名称:

```
ALTER VIEW myview RENAME TO product_view;
```

修改视图所属schema:

```
ALTER VIEW product_view SET schema public;
```

重建视图：

```
ALTER VIEW public.product_view REBUILD;
```

重建依赖视图：

```
ALTER VIEW ONLY public.product_view REBUILD;
```

相关链接

CREATE VIEW，**DROP VIEW**

12.28 CLEAN CONNECTION

功能描述

当数据库有异常时，用来清理数据库连接。允许在指定节点上清理指定数据库的指定用户的相关连接。

注意事项

无

语法格式

```
CLEAN CONNECTION
   TO { COORDINATOR ( nodename [, ... ] ) | NODE ( nodename [, ... ] )| ALL [ CHECK ] [ FORCE ] }
   [ FOR DATABASE dbname ]
   [ TO USER username ];
```

参数说明

- **CHECK**

  仅在节点列表为TO ALL时可以指定。如果指定该参数，会在清理连接之前检查数据库是否被其他会话连接访问。此参数主要用于DROP DATABASE之前的连接访问检查，如果发现有其他会话连接，则将报错并停止删除数据库。

- **FORCE**

  仅在节点列表为TO ALL时可以指定，如果指定该参数，所有与指定dbname和username相关的线程都会收到SIGTERM信号，然后被强制关闭。

- **COORDINATOR (nodename [, ...]) | NODE (nodename [, ...]) | ALL**

  删除指定节点上的连接。有三种场景：

  - 删除指定CN上的连接。
  - 删除指定DN上的连接。
  - 删除所有节点上的连接，包括CN和DN。

  取值范围：可替换其中的nodename为已存在的节点名。

- **dbname**

  删除指定数据库上的连接。如果不指定，则删除所有数据库的连接。

  取值范围：已存在数据库名。

- **username**

  删除指定用户上的连接。如果不指定，则删除所有用户的连接。

  取值范围：已存在的用户。

  说明

  参数**dbname**和**username**必须至少指定一个。

示例

删除数据库template1在dn1和dn2节点上的连接：

```
CLEAN CONNECTION TO NODE (dn1,dn2) FOR DATABASE template1;
```

删除用户jack在dn1节点上的连接。

```
CLEAN CONNECTION TO NODE (dn1) TO USER jack;
```

删除在数据库gaussdb上的所有连接。

```
CLEAN CONNECTION TO ALL FORCE FOR DATABASE gaussdb;
```

12.29 CLOSE

功能描述

CLOSE用于释放和一个游标关联的所有资源。

注意事项

- 不允许对一个已关闭的游标再做任何操作。
- 一个不再使用的游标应该尽早关闭。
- 当创建游标的事务用COMMIT或ROLLBACK终止之后，每个不可保持的已打开游标都隐含关闭。
- 当创建游标的事务通过ROLLBACK退出之后，每个可以保持的游标都将隐含关闭。
- 当创建游标的事务成功提交，可保持的游标将保持打开，直到执行一个明确的CLOSE或者客户端断开。
- GaussDB(DWS)没有明确打开游标的OPEN语句，因为一个游标在使用CURSOR命令定义的时候就打开了。可以通过查询系统视图pg_cursors看到所有可用的游标。

语法格式

```
CLOSE { cursor_name | ALL } ;
```

参数说明

- **cursor_name**

  一个待关闭的游标名称。

- **ALL**

  关闭所有已打开的游标。

示例

关闭游标：

```
CLOSE ALL;
```

相关链接

FETCH，**MOVE**

12.30 CLUSTER

功能描述

根据一个索引对表进行聚簇排序。

CLUSTER指示GaussDB(DWS)基于索引名指定的索引来聚簇由表名指定的表。 索引名指定的索引该索引必须已经定义在指定表上。

当对一个表聚簇后，该表将基于索引信息进行物理排序。聚簇是一次性操作：当表被更新之后， 更改的内容不会被聚簇。也就是说，系统不会试图按照索引顺序对新的存储内容及更新记录进行重新聚簇。

在对一个表聚簇之后，GaussDB(DWS)会记录在哪个索引上建立了聚簇。 形式CLUSTER table_name会使用前面所用的同一个索引对表重新聚簇。用户也可以用CLUSTER或ALTER TABLE的SET WITHOUT CLUSTER形式把索引设置为可用于后续的聚簇操作或清除任何之前的设置。

不含参数的CLUSTER会将当前用户所拥有的数据库中的先前做过聚簇的所有表重新聚簇，如果是系统管理员调用，则是所有已被聚簇过的表。

在对一个表进行聚簇的时候，会在其上请求一个ACCESS EXCLUSIVE锁。这样就避免了在CLUSTER完成之前对此表执行其它的操作(包括读写)。

注意事项

- 只有行存B-tree索引支持CLUSTER操作。
- 如果用户只是随机访问表中的行，那么表中数据的实际存储顺序是无关紧要的。但是， 如果对某些数据的访问多于其它数据，而且有一个索引将这些数据分组，那么使用CLUSTER中会有所帮助。如果从一个表中请求一定索引范围的值， 或者是一个索引值对应多行，CLUSTER也会有所帮助，因为如果索引标识出第一匹配行所在的存储页，所有其它行也可能已经在同一个存储页里了，这样便节省了磁盘访问的时间，加速了查询。
- 在聚簇过程中，系统创建一个按照索引顺序建立的表的临时复制的同时，也会建立表上的每个索引的临时复制。因此，需要磁盘上有足够的剩余空间， 至少是表大小和索引大小的和。
- 因为CLUSTER会记住聚集信息，可以第一次手动聚簇想要聚簇的表，然后设置一个定期运行的维护脚本，其中执行不带任何参数的CLUSTER，这样那些表就会被周期性地重新聚簇。
- 因为优化器记录着有关表的排序的统计，所以建议在新近聚簇的表上运行ANALYZE。否则，优化器可能会选择很差劲的查询规划。
- CLUSTER不允许在事务中执行。
- 对表执行CLUSTER操作时会触发表重建（表重建过程中会先把数据转储到一个新的数据文件中，重建完成之后会删除原始文件），当表比较大时，重建会消耗较多的磁盘空间。当磁盘空间不足时，要谨慎对待大表CLUSTER操作，防止触发集群只读。

语法格式

- 对一个表进行聚簇排序。

```
CLUSTER [ VERBOSE ] table_name [ USING index_name ];
```


- 对一个分区进行聚簇排序。
  ```
  CLUSTER [ VERBOSE ] table_name PARTITION ( partition_name ) [ USING index_name ];
  ```
- 对已做过聚簇的表重新进行聚簇。
  ```
  CLUSTER [ VERBOSE ];
  ```

参数说明

- **VERBOSE**

  启用显示进度信息。
- **table_name**

  表名称。

  取值范围：已存在的表名称。
- **index_name**

  索引名称。

  取值范围：已存在的索引名称。
- **partition_name**

  分区名称。

  取值范围：已存在的分区名称。

示例

创建一个分区表：

```
CREATE TABLE inventory_p1
(
   INV_DATE_SK              INTEGER              NOT NULL,
   INV_ITEM_SK              INTEGER              NOT NULL,
   INV_WAREHOUSE_SK         INTEGER              NOT NULL,
   INV_QUANTITY_ON_HAND     INTEGER
)
DISTRIBUTE BY HASH(INV_ITEM_SK)
PARTITION BY RANGE(INV_DATE_SK)
(
      PARTITION P1 VALUES LESS THAN(2451179),
      PARTITION P2 VALUES LESS THAN(2451544),
      PARTITION P3 VALUES LESS THAN(2451910),
      PARTITION P4 VALUES LESS THAN(2452275),
      PARTITION P5 VALUES LESS THAN(2452640),
      PARTITION P6 VALUES LESS THAN(2453005),
      PARTITION P7 VALUES LESS THAN(MAXVALUE)
);
```

创建索引ds_inventory_p1_index1。

```
CREATE INDEX ds_inventory_p1_index1 ON inventory_p1 (INV_ITEM_SK) LOCAL;
```

对表inventory_p1进行聚集：

```
CLUSTER inventory_p1 USING ds_inventory_p1_index1;
```

对分区p3进行聚集：

```
CLUSTER inventory_p1 PARTITION (p3) USING ds_inventory_p1_index1;
```

对数据库中可以进行聚集的表进聚集：

```
CLUSTER;
```


12.31 COMMENT

功能描述

定义或修改对象的注释。

注意事项

- 每个对象只存储一条注释，因此要修改对象的注释，对同一个对象发出一条新的COMMENT命令即可。要删除注释，在文本字符串的位置写上NULL即可。当删除对象时，注释自动被删除掉。
- 目前注释浏览没有安全机制：任何连接到数据库上的用户都可以看到所有该数据库对象的注释。共享对象（比如数据库、角色、表空间）的注释是全局存储的，连接到任何数据库的任何用户都可以看到它们。因此，不要在注释里存放与安全有关的敏感信息。
- 对大多数对象来说，只有对象的所有者可以设置注释。角色没有所有者，所以COMMENT ON ROLE命令仅可以由系统管理员对系统管理员角色执行，有CREATE ROLE权限的角色也可以为非系统管理员角色设置注释。系统管理员可以对所有对象进行注释。

语法格式

```
COMMENT ON
{
  AGGREGATE agg_name (agg_type [, ...] ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  COLUMN { table_name.column_name | view_name.column_name } |
  CONSTRAINT constraint_name ON table_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ {[ argmode ] [ argname ] argtype} [, ...] ] ) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SERVER object_name |
  TABLE object_name |
  TABLESPACE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TYPE object_name |
  VIEW object_name
}
   IS 'text';
```

参数说明

- **agg_name**
  聚集函数的名称。
- **agg_type**
  聚集函数参数的类型。
- **source_type**
  类型转换的源数据类型。
- **target_type**
  类型转换的目标数据类型。
- **object_name**
  对象名。
- **table_name.column_name/view_name.column_name**
  定义/修改注释的列名称。前缀可加表名称或者视图名称。
- **constraint_name**
  定义/修改注释的表约束的名称。
- **table_name**
  表的名称。
- **function_name**
  定义/修改注释的函数名称。
- **argmode,argname,argtype**
  函数参数的模式、名称、类型。
- **large_object_oid**
  定义/修改注释的大对象的OID值。
- **operator_name**
  操作符名称。
- **left_type,right_type**
  操作参数的数据类型（可以用模式修饰）。当前置或者后置操作符不存在时，可以增加NONE选项。
- **text**
  注释。

示例

创建表：

```
CREATE TABLE IF NOT EXISTS CUSTOMER
(
   C_CUSTKEY     BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY ,
   C_NAME        VARCHAR(25) ,
   C_ADDRESS     VARCHAR(40) ,
   C_NATIONKEY   INT         ,
   C_PHONE       CHAR(15)    ,
   C_ACCTBAL     DECIMAL(15,2)
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

为customer.c_customer_sk列加注释：


```
COMMENT ON COLUMN customer.C_CUSTKEY IS 'Primary key of customer demographics table.';
```

创建视图：

```
CREATE OR REPLACE VIEW customer_details_view AS SELECT * FROM CUSTOMER;
```

为customer_details_view视图加注释：

```
COMMENT ON VIEW customer_details_view IS 'View of customer detail';
```

为customer表加注释：

```
COMMENT ON TABLE customer IS 'This is my table';
```

12.32 CREATE BARRIER

功能描述

创建一个新集群节点间的同步点。该同步点可用于数据恢复。

注意事项

在创建同步点之前，首先要确认集群中CN和DN的gtm_backup_barrier和enable_cbm_tracking参数都已设置为on。

语法格式

```
CREATE BARRIER [ barrier_name  ] ;
```

参数说明

barrier_name

可选参数。同步点名称。

取值范围：字符串，要符合标识符的命名规范。

示例

创建一个同步点，不指定名称：

```
CREATE BARRIER;
```

创建一个同步点并指定其名称为barrier1：

```
CREATE BARRIER 'barrier1';
```

12.33 CREATE DATABASE

功能描述

创建一个新的数据库。缺省情况下新数据库将通过复制标准系统数据库template1来创建。可以通过TEMPLATE template指定不同的模板。

注意事项

- 只有拥有CREATEDB权限的用户才可以创建新数据库，系统管理员默认拥有此权限。
- 不能在事务块中执行创建数据库语句。
- 在创建数据库过程中，若出现类似“could not initialize database directory”的错误提示，可能是由于文件系统上数据目录的权限不足或磁盘满等原因引起。

语法格式

```
CREATE DATABASE database_name
   [ [ WITH ] { [ OWNER [=] user_name ] |
            [ TEMPLATE [=] template ] |
            [ ENCODING [=] encoding ] |
            [ LC_COLLATE [=] lc_collate ] |
            [ LC_CTYPE [=] lc_ctype ] |
            [ DBCOMPATIBILITY [=] compatibilty_type ] |

            [ CONNECTION LIMIT [=] connlimit ]}[...] ];
```

参数说明

- **database_name**

  数据库名称。

  取值范围：字符串，要符合标识符的命名规范。

- **OWNER [=] user_name**

  数据库所有者。缺省时，新数据库的所有者是当前用户。

  取值范围：已存在的用户名。

- **TEMPLATE [=] template**

  模板名。即从哪个模板创建新数据库。GaussDB(DWS)采用从模板数据库复制的方式来创建新的数据库。初始时，GaussDB(DWS)包含两个模板数据库template0、template1，以及一个默认的用户数据库gaussdb。

  取值范围：已有数据库的名称。不指定时，系统默认复制template1。另外，不支持指定为gaussdb数据库。

  须知

  目前不支持模板库中含有SEQUENCE对象。如果模板库中有SEQUENCE，则会创建数据库失败。

- **ENCODING [=] encoding**

  指定数据库使用的字符编码，可以是字符串（如'SQL_ASCII'）、整数编号。

  不指定时，默认使用模版数据库的编码。模板数据库template0和template1的编码默认与操作系统环境相关。template1不允许指定字符编码，因此若要创建数据库时指定字符编码，请使用template0创建数据库。即如果需要指定encoding，需要和template参数搭配使用，且template取值为template0。

  常用取值：GBK、UTF8、Latin1。

须知

- 可使用“show server_encoding;”命令查看当前数据库的字符编码集。
- 为了适应全球化的需求，使数据库编码能够存储与表示绝大多数的字符，建议创建Database的时候使用UTF8编码。
- 指定新的数据库字符集编码必须与所选择的本地环境中（LC_COLLATE和LC_CTYPE）的设置兼容。
- 当指定的字符编码集为GBK时，部分中文生僻字无法直接作为对象名。这是因为GBK第二个字节的编码范围在0x40-0x7E之间时，字节编码与ASCII字符@A-Z[\]^_`a-z{|}重叠。其中@[\]^_'{|}是数据库中的操作符，直接作为对象名时，会语法报错。例如“侅”字，GBK16进制编码为0x8240，第二个字节为0x40，与ASCII“@”符号编码相同，因此无法直接作为对象名使用。如果确实要使用，可以在创建和访问对象时，通过增加双引号来规避这个问题。
- 当前版本GBK字符集支持了欧元符'€'，十六进制表示为'0x80'，用户可以在GBK库中操作欧元符，也使得GaussDB(DWS)的GBK字符集可以兼容CP936字符集。需注意GBK字符集约等于CP936字符集，但是GBK字符集中不包含欧元符的定义。

- **LC_COLLATE [=] lc_collate**

  指定新数据库使用的字符集。例如，通过lc_collate = 'zh_CN.gbk'设定该参数。

  该参数的使用会影响到对字符串的排序顺序（如使用ORDER BY执行，以及在文本列上使用索引的顺序）。默认是使用模板数据库的排序顺序。若要创建数据库时指定字符集，请使用template0创建数据库。即如果需要指定encoding，需要和template参数搭配使用，且template取值为template0。

  取值范围：有效的排序类型。

- **LC_CTYPE [=] lc_ctype**

  指定新数据库使用的字符分类。例如，通过lc_ctype = 'zh_CN.gbk'设定该参数。该参数的使用会影响到字符的分类，如大写、小写和数字。默认是使用模板数据库的字符分类。若要创建数据库时指定字符分类，请使用template0创建数据库。即如果需要指定encoding，需要和template参数搭配使用，且template取值为template0。

  取值范围：有效的字符分类。

- **DBCOMPATIBILITY [=] compatibilty_type**

  指定兼容的数据库的类型。

  取值范围：ORA、TD、MySQL。分别表示兼容Oracle、Teradata和MySQL数据库。若不指定该参数，默认为ORA。

- **CONNECTION LIMIT [=] connlimit**

  数据库可以接受的并发连接数。

  取值范围：>=-1的整数。默认值为-1，表示没有限制。

须知

- 系统管理员不受此参数的限制。
- 为保证集群正常使用，connection limit的最小值是集群中CN的数目。在集群做ANALYZE时，其他CN节点会连接当前做ANALYZE的CN节点来同步元数据。例如集群中有3个CN节点，那么connection limit应该设置为>=3。

有关字符编码的一些限制：

- 若区域设置为C（或POSIX），则允许所有的编码类型，但是对于其他的区域设置，字符编码必须和区域设置相同。
- 编码和区域设置必须匹配模板数据库，除了将template0当作模板。 因为其他数据库可能会包含不匹配指定编码的数据，或者可能包含排序顺序受LC_COLLATE和LC_CTYPE影响的索引。复制这些数据会导致在新数据库中的索引失效。template0是不包含任何会受到影响的数据或者索引。
- 支持的有效编码类型与当前所处的环境有关。若出现“invalid locale name”的字可通过locale -a命令检查环境所支持的字符编码集。

示例

创建一个GBK编码的数据库music（本地环境的编码格式必须也为GBK）：

```
CREATE DATABASE music ENCODING 'GBK' template = template0;
```

创建数据库music2，并指定所有者为jim：

```
CREATE DATABASE music2 OWNER jim;
```

用模板template0创建数据库music3，并指定所有者为jim：

```
CREATE DATABASE music3 OWNER jim TEMPLATE template0;
```

创建兼容ORA格式的数据库：

```
CREATE DATABASE ora_compatible_db DBCOMPATIBILITY 'ORA';
```

相关链接

ALTER DATABASE，**DROP DATABASE**

12.34 CREATE FOREIGN TABLE (GDS 导入导出)

创建GDS外表。

功能描述

在当前数据库创建一个GDS外表，用于数据并行导入导出。GDS外表分为只读外表和只写外表，分别用于数据并行导入和并行导出，缺省为只读外表。

注意事项

- 外表由命令执行者所有；
- GDS外表不需要显式指定分布方式，默认支持ROUNDROBIN分布方式；
- 对于GDS外表指定任何约束（列约束、表约束等）均不生效。
- GDS导入导出支持的文件格式：TEXT、CSV和FIXED。

语法格式

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
   ( [ { column_name type_name POSITION(offset,length) | LIKE source_table } [, ...] ] )
   SERVER gsmpp_server
   OPTIONS ( { option_name ' value ' } [, ...] )
   [ { WRITE ONLY | READ ONLY }]
```

```
[ WITH error_table_name | LOG INTO error_table_name]
[REMOTE LOG 'name']
[PER NODE REJECT LIMIT 'value']
[ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ];
```

参数概览

创建外表语法提供了多个参数，常用参数分类如下。

- 必需参数
  - **table_name**
  - **column_name**
  - **type_name**
  - **SERVER gsmpp_server**
  - **OPTIONS可选参数**
    - 外表的数据源位置参数**location**
    - 数据格式参数
      - **format**
      - **header**（仅支持CSV，FIXED格式）
      - **fileheader**（仅支持CSV，FIXED格式）
      - **out_filename_prefix**
      - **delimiter**
      - **quote**（仅支持CSV格式）
      - **escape**（仅支持CSV格式）
      - **null**
      - **noescaping**（仅支持TEXT格式）
      - **encoding**
      - **eol**
      - **conflict_delimiter**
      - **file_type**
      - **auto_create_pipe**
      - **del_pipe**
    - 容错性参数
      - **fill_missing_fields**
      - **ignore_extra_data**
      - **reject_limit**
      - **compatible_illegal_chars**
    - 性能参数
      - **file_sequence**
- 可选参数

  - **WITH error_table_name**
  - **LOG INTO error_table_name**
  - **REMOTE LOG 'name'**
  - **PER NODE REJECT LIMIT 'value'**

参数说明

- **IF NOT EXISTS**

  如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。

- **table_name**

  外表的表名。

  取值范围：字符串，要符合标识符的命名规范。

- **column_name**

  外表中的字段名。

  取值范围：字符串，要符合标识符的命名规范。

- **type_name**

  字段的数据类型。

- **POSITION(offset,length)**

  在固定长度模式中，定义每一个字段在数据文件中的位置。

  说明

  offset为该字段在数据源文件中的起始位置，length为该字段的长度。

  取值范围：offset取值不能小于0，单位为字节。

  每条记录的长度不能大于1GB，文件中没有出现的列默认以空值代替。

- **SERVER gsmpp_server**

  外表的server名字。对于GDS外表，其server是初始数据库默认创建的，即gsmpp_server。

- **OPTIONS ({ option_name ' value ' } [, ...])**

  用于指定外表数据的各类参数。

  - location

    外表的数据源位置，目前支持URL方式的描述。多个URL使用‘|’分隔。

    gds目前可以支持导出的时候自动创建外表定义的目录。 如外表localtion 定义 "gsfs:// 192.168.0.91:5000/2019/09" 执行导出任务的时候，如果gds数据目录下的子目录 "2019/09" 不存在则会自动的创建该子目录，不需要用户手动创建外表中指定的目录。

说明

- 对于使用GDS从远端服务器并行导入时的只读外表（默认为只读）的URL末尾必须指定文件的匹配模式或者文件名。

  例如: gsfs://192.168.0.90:5000/*或者file:///data/data.txt或者gsfs://192.168.0.90:5000/* | gsfs:// 192.168.0.91:5000/*

- 对于使用GDS并行导出到远端服务器时的只写外表，URL不需要指定文件名。当导出数据文件存储位置为远端URL时，例如gsfs:// 192.168.0.90:5000/，则数据源位置可指定多个，此时：若导出数据文件存储位置数量小于等于数据节点数量时，使用此外部表执行导出任务，数据将被平均分配至各数据源位置；若导出数据存储位置数量大于数据节点数量时，执行导出任务，数据将被平均分配给此位置列表中从前端开始等于数据节点数量的数据源位置下，剩余数据源位置仍会创建数据文件，但文件中不会有任何数据。
- 对于使用GDS从远端服务器并行导入时的只读外表，URL个数应小于DN个数，且不能使用多个location相同的URL。
- 当使用gsfss协议，即当URL为“gsfss://”开头，进行加密导入导出时，并发数量不能超过10。
- gds导出时localtion指定的文件路径“gsfs://127.0.0.1:7789/2019/09/”中的2019/09子目录会在执行导出任务的时候自动创建。
- 设置file_type为“pipe”时，GDS会根据URL中最后一个字符是否为“/”来判断导入导出的目标是管道文件还是目录。如：
  - gsfs://192.168.0.90:5000/a/b，GDS会将b识别成一个管道文件。
  - gsfs://192.168.0.90:5000/a/b/，GDS将b识别成一个目录，并在b目录下创建管道文件。

– format

外表中数据源文件的格式。

取值范围：CSV、TEXT、FIXED，缺省值为TEXT。

- CSV格式的文件，对一些转义序列按照普通字符串进行处理，因此可以有效处理数据列中的换行符。
- TEXT格式的文件，可以有效处理一些转义序列，因此无法正确处理数据列中的换行符。
- FIXED格式的文件，适用于每条数据的数据列都比较固定的数据，长度不足的列会添加空格补齐。

说明

- 转义序列指的是反斜杠开头的字符串，包括：\b（退格）、\f（换页）、\n（换行）、\r（回车）、\t（横向制表）、\v（纵向制表）、\数字（八进制编码）、\x数字（十六进制编码）。TEXT格式可以按照本身含义进行处理，其他格式只能按照普通字符串进行处理。
- 定长格式（FIXED）定义如下。当为FIXED时，必须为每一列指定POSITION。
  1. 每条记录的每个字段长度相同。
  2. 长度不足的字段以空格填充，数字类型字段左对齐，字符字段右对齐。
  3. 字段和字段之间没有分隔符。

– header

指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。header只能用于CSV，FIXED格式的文件中。

在导入数据时，如果header选项为on，则数据文本第一行会被识别为标题行，会忽略此行。如果header为off，而数据文件中第一行会被识别为数据。

在导出数据时，如果header选项为on，则需要指定**fileheader**。fileheader用来指定导出头文件的格式。如果header为off，则导出数据文件不包含标题行。

取值范围：true/on，false/off。缺省值为false/off。

- fileheader

  指定导出数据要包含的标题行定义的文件，文件一般只包含一行用来描述每一列数据信息的字符串。

  例如：在包含商品信息的数据前加标题行，定义文件如下

  The information of products.\n

  须知

  - 标题行定义文件仅在header为on或true的情况下有效，且需要提前写好备用。
  - 在Remote导出模式下，定义文件必须放在GDS的工作目录（即启动gds时指定的-d路径）下。
  - 定义文件只能包含一行标题信息，并以换行符结尾，多余的行将被丢弃（标题信息不能包含换行符）。
  - 定义文件包括换行符在内长度不超过1M。

- out_filename_prefix

  指定write only外表导出时，GDS端生成导出数据文件的文件名前缀。

  file_type设置为pipe时，会生成“dbName_schemaName_foreignTableName.pipe”的管道文件。

  如果out_filename_prefix和location中都指定了管道名，则以location中指定的管道文件名为准。

须知

- 指定文件名前缀需合法，符合GDS部署物理环境使用的文件系统的约束，否则出现文件创建失败：
  - 指定的导出文件名前缀中不含有非法字符，其中非法字符包含但不限于'/', '?', '*', ':', '|', '\\', '<', '>', '@', '#', '$', '&', '(', ')', '+', '-'，允许的字符范围为[a-z]*[A-Z]*[0-9]*和'_'
  - 指定的导出文件名前缀中不可以是一些Windows和linux预留的特性字段，其中包括但不限于：

    "con","aux","nul","prn","com0","com1","com2","com3","com4","com5","com6","com7","com8","com9","lpt0","lpt1","lpt2","lpt3","lpt4","lpt5","lpt6","lpt7","lpt8","lpt9"
  - 指定的导出文件名前缀，与GDS -d目录和“.dat”或者“.pipe”拼接为绝对路径后必须符合GDS所在部署文件系统的文件名长度要求。
  - 指定的导出文件名前缀，需要可以被数据文件的最终接收方正确解析识别（包括但不限于GDS再次导入库中），对于造成文件名解析问题的指定选项，需要用户识别。
- 在多文件同时导出的高并发导出场景下，请确认并发的导出任务不要使用同一个文件名前缀设定，否则从操作系统/文件系统层面可能会出现导出文件的覆盖和丢失。

- delimiter

  指定数据文件行数据的字段分隔符，不指定则使用默认分隔符，TEXT格式的默认分隔符是水平制表符（tab），CSV格式的默认分隔符为“,”，FIXED格式没有分隔符。

  说明

  - 分隔符不能是\r和\n。
  - 分隔符不能和null参数相同，CSV格式数据的分隔符不能和quote参数相同。
  - TEXT格式数据的分隔符不能包含：\.abcdefghijklmnopqrstuvwxyz0123456789。
  - 数据文件中单行数据长度需<1GB，如果分隔符较长且数据列较多的情况下，会影响导出有效数据的长度。
  - 分隔符推荐使用多字符（例如'$^&'）和不可见字符（例如0x07、0x08、0x1b等）。
  - 建议TEXT格式下多字符分隔符中的字符不要完全相同，例如不建议使用 delimiter '---' 。

  取值范围：

  支持多字符分隔符，但分隔符不能超过10个字节。

- quote

  用于设置将CSV格式数据源文件中的什么字符识别为引号字符。缺省值为双引号。

  说明

  - quote参数不能和分隔符、null参数相同。
  - quote参数只能是单字节的字符。
  - 推荐不可见字符作为quote，例如0x07，0x08，0x1b等。

- escape

  用来指定CSV格式的数据源文件中，什么字符为逃逸字符。逃逸字符只能指定为单字节字符。

  缺省值和quote相同。

- null

  用来指定数据文件中空值的表示。

  说明

  - null值不能是\r和\n，最大为100个字符。
  - null值不能和分隔符、quote参数相同。

  取值范围：

  - 在TEXT格式下缺省值是\n。
  - CSV格式下缺省值是一个没有引号的空字符串。

- noescaping

  TEXT格式下，不对'\'和后面的字符进行转义。

  说明

  noescaping参数只在TEXT格式下有效。

  取值范围：true/on，false/off。缺省值为false/off。

- encoding

  指定数据文件的编码格式名称，即需要以何编码格式对数据文件进行解析和校验/输出文件为何种编码格式。缺省值为当前数据库的默认客户端编码格式，即client_encoding。

  导入外表此处强烈建议指定为文件的编码格式，或根据文件的字符集在导入前对client_encoding进行设置。否则可能会导致不必要的解析、校验错误以及其导致的导入报错回滚，甚至非法数据入库。导出外表同样希望指定此选项，以避免导出采用默认字符集设置时与预期不符。

  在创建外表时此选项未指定，会在客户端给出对应Warning信息。

  说明

  目前GDS导入外表不支持解析带有多种字符集编码格式混合的文件，GDS导出外表不支持写出带有多种字符集编码格式混合的文件。

- fill_missing_fields

  当数据导入时，若数据源文件中一行的最后一个字段缺失的处理方式。

  取值范围：true/on，false/off。缺省值为false/off。

  - 参数为true/on，当数据导入时，若数据源文件中一行数据的最后一个字段缺失，则把最后一个字段的值设置为NULL，不报错。
  - 参数为false/off，如果最后一个字段缺失会显示如下错误信息。

    ```
    missing data for column "tt"
    ```

- ignore_extra_data

  若数据源文件比外表定义列数多，是否会忽略对多出的列。该参数只在数据导入过程中使用。

  取值范围：true/on，false/off。缺省值为false/off。

- 参数为true/on，若数据源文件比外表定义列数多，则忽略行尾多出来的列。
- 参数为false/off，若数据源文件比外表定义列数多，会显示如下错误信息。
  ```
  extra data after last expected column
  ```

须知

如果行尾换行符丢失，使两行变成一行时，设置此参数为true将导致后一行数据被忽略掉。

- reject_limit

  指定本次数据导入允许出现的数据格式错误个数，当导入过程中出现的数据格式错误未达到限定值时，本次数据导入可以成功。

须知

此语法建议用PER NODE REJECT LIMIT 'value'替代。

数据格式错误是指缺少或者多出字段值，数据类型错误或者编码错误。对于非数据格式错误，一旦发生就将导致整个数据导入失败。

取值范围：正整型值、unlimited（无限制）。

不指定该参数时，有错误信息立即返回。

说明

指定正整型参数时需要添加单引号。

- mode

  指定数据导入过程中，数据导入策略。GaussDB(DWS)只支持Normal策略。

  取值范围：

  - Normal（缺省值）：支持所有文件格式(包括CSV、TEXT、FIXED)，数据导入需要在数据服务器上启动Gauss data service协助完成。

- eol

  指定导入导出数据文件换行符样式。

  取值范围：支持多字符换行符，但换行符不能超过10个字节。常见的换行符，如\r、\n、\r\n（设成0x0D、0x0A、0x0D0A效果是相同的），其他字符或字符串，如$、#。

  说明

  - eol参数只能用于TEXT格式的导入导出，不支持CSV格式和FIXED格式导入。为了兼容原有eol参数，仍然支持导出CSV格式和FIXED格式时指定eol参数为0x0D或0x0D0A。
  - eol参数不能和分隔符、null参数相同。
  - eol参数不能包含：数字，字母和符号“.”。

- conflict_delimiter

此参数一般配合compatible_illegal_chars参数一起使用，当用户的数据文件中包含半个汉字字符的时候，并且这半个字符和分隔符会由于外表的编码和数据库编码不一致被编码成一个其他的汉字，导致分隔符被掩盖从而报错缺少字段。

如果用户不希望让这半个字符和分隔符编码成一个其他字符则可以使用此参数。

取值范围：true/on，false/off。缺省值为false/off。

- 参数为true/on，允许这半个字符和分隔符编码成一个其他字符。
- 参数为false/off，不允许这半个字符和分隔符编码成一个其他字符。

须知

此参数默认关闭。由于场景较少出现，不建议打开。如果未能识别场景而打开此参数则会有入表信息错乱的风险。

例如：假设有一行GBK数据"3|+|腰+|20191212"要被导入到UTF8的数据库，用户自定义的字段分隔符为"|+|"。

这行GBK数据和十六进制对应关系如下

33 7C 2B 7C C4 7C 2B 7C 323031393132313

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

3 | + | ** | + | 20191212

当导入UTF8数据库时，数据库会发生如下转换。

GBK -> UTF8

33->33 (3)

7C->7C (|)

2B->2B (+)

7C->7C(|)

C47C-> E886A2(腰)

2B->2B(+)

7C->7C(|)

323031393132313-> 20191212

由于用户定义的字段分隔符号为"|+|"，因此对该行数据分隔后获取的数据是3，腰+|20191212,

实际上用户可能要获得的是3，？,20191212。

"腰" 这个字符其实是一个用户不期望的半个字符，和用户定义的分隔符“|+|”中的“|”产生了冲突，导致数据错乱或者导入失败。

如果用户想要忽略这种字符可以使用conflict_delimiter配合compatible_illegal_chars参数将“C4”这个半个GBK字符准换成“？”导入UTF8数据库。

- file_type

  指定导入或者导出的文件类型。

  取值范围：normal，pipe。缺省为normal。

- 参数为normal，表示导入或者导出的文件类型为普通文件。
- 参数为pipe，表示导入或者导出的文件类型为命名管道文件。

– file_sequence

用于多任务GDS外表并行导入，提升单个文件的导入性能。该参数仅供数据导入使用。

格式为file_sequence '文件被拆分的总数-当前分片' 。例如：

file_sequence '3-1' 表示导入的文件在逻辑上被拆分成3份，当前外表导入的数据为第一个分片上的数据。

file_sequence '3-2' 表示导入的文件在逻辑上被拆分成3份，当前外表导入的数据为第二个分片上的数据。

file_sequence '3-3' 表示导入的文件在逻辑上被拆分成3份，当前外表导入的数据为第三个分片上的数据。

使用该参数需遵循以下约束：

- 文件被拆分的总数小于等于8。
- 当前分片小于等于文件被拆分的总数。
- 导入的文件仅支持CSV和text格式。

说明

使用CSV格式进行并行导入时，在如下示例场景中会因为CSV本身的规则和GDS拆分逻辑冲突而导致其中的某些分片导入失败。

场景：csv文件中包含未转义的换行符，且该换行符被quote指定的字符所包含，并且该行数据处于逻辑分片的第一行。

示例：并行导入一个文件big.csv ，正确导入显示内容如下：

```
--id, username, address
10001,"customer1 name","Rose District"
10002,"customer2 name","
23 Road Rose
District NewCity"
10003,"customer3 name","NewCity"
```

文件被拆分成两份后，第一个分片显示内容如下：

```
10001,"customer1 name","Rose District"
10002,"customer2 name","
23
```

第二个分片显示内容如下：

```
Road Rose
District NewCity"
10003,"customer3 name","NewCity"
```

因为第二个分片第一行后面的换行符包含在一个双引号之间，导致GDS无法分辨该换行符是字段中的换行符还是行中的分隔符，因此第一个分片会成功导入两条数据，第二个分片导入失败。

– auto_create_pipe

用于设置GDS进程是否自动创建命名管道文件。

取值范围：true/on，false/off。缺省值为true/on。

- 参数为true/on，表示允许GDS进程自动创建命名管道文件。
- 参数为false/off，表示用户需手动创建命名管道文件。

须知

- 设置auto_create_pipe参数时，file_type必须设置为pipe，否则不能成功创建外表。
- auto_create_pipe设置为false，执行导入导出时，若未指定管道文件，会打开“数据库名_模式名_外表名.pipe”文件；若已指定管道文件，会打开location参数中指定的管道文件。该命名管道文件在pipe-timeout参数设置的时间内没有被其他程序写入或者以写的方式打开，则导入导出任务报错超时。若发现该文件不是管道文件则导入导出任务直接报错。
- auto_create_pipe设置为true，执行导入导出时，若未指定管道文件，会打开“数据库名_模式名_外表名.pipe”文件；若该文件为普通文件类型，则导入导出任务报错。若为管道文件会自动删除该文件，并重新创建该命名管道文件。
- 管道文件导出时**location**参数可以指定导出的管道文件，如：location‘gsfs://127.0.0.1:7789/aa.pipe'，当auto_create_pipe设置为true，GDS会自动在数据目录下创建“aa.pipe”管道文件。

– del_pipe

用于设置不落地导入/导出业务结束后是否自动删除管道文件。

取值范围：true/on，false/off。缺省值为true/on。

- 参数为true/on，表示允许GDS进程自动删除命名管道文件。
- 参数为false/off，表示GDS进程不会删除命名管道文件。

须知

设置del_pipe时，file_type必须设置为pipe，否则不能成功创建外表。

– fix

指定每一行定长格式数据的长度。按字节计算。此语法仅对READ ONLY的外表有效。

取值范围：N >= POSITION指定的总长度（总长度即为表定义最后一个字段的offset与length的和）AND N < 1GB

– out_fix_alignment

指定定长导出中，指定BYTEAOID、CHAROID、NAMEOID、TEXTOID、BPCHAROID、VARCHAROID、NVARCHAR2OID、CSTRINGOID对应类型所在列的对齐方式。

取值范围：align_left、align_right

默认值：align_right

须知

由于bytea数据类型要求十六进制格式（如"\XXXX"）或八进制格式（如"\XXX\XXX\XXX"），导入时需要左对齐（即列数据以两种格式开头，而非空格）。因此若导出文件需要重新以GDS外表入库且数据长度小于外表formatter指定长度，导出时需要指定左对齐，否则会在入库的过程中报错。

- date_format

  导入对于DATE类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法DATE格式。可参考**时间、日期处理函数和操作符**。

  说明

  对于指定为ORACLE兼容类型的数据库，则DATE类型内建为TIMESTAMP类型。在导入的时候，若需指定格式，可以参考下面的timestamp_format参数。

- time_format

  导入对于TIME类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法TIME格式，不支持时区。可参考**时间、日期处理函数和操作符**。

- timestamp_format

  导入对于TIMESTAMP类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法TIMESTAMP格式，不支持时区。可参考**时间、日期处理函数和操作符**。

- smalldatetime_format

  导入对于SMALLDATETIME类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法SMALLDATETIME格式。可参考**时间、日期处理函数和操作符**。

- compatible_illegal_chars

  导入非法字符容错参数。此语法仅对READ ONLY的外表有效。

  取值范围：true/on，false/off。缺省值为false/off。

  - 参数为true/on，则导入时遇到非法字符进行容错处理，非法字符转换后入库，不报错，不中断导入。
  - 参数为false/off，导入时遇到非法字符进行报错，中断导入。

  说明

  导入非法字符容错规则如下：

  （1）对于'\0'，容错后转换为空格；

  （2）对于其他非法字符，容错后转换为问号；

  （3）如果compatible_illegal_chars为true/on标识导入时对于非法字符进行容错处理，如果NULL、DELIMITER、QUOTE、ESCAPE设置为空格或问号则会通过如"illegal chars conversion may confuse COPY escape 0x20"等报错信息提示用户修改可能引起混淆的参数，以避免导入错误。

- **READ ONLY**

  只读外表，该参数只供数据导入使用。

- **WRITE ONLY**

  只写外表，该参数只供数据导出使用。

- **WITH error_table_name**

  数据导入过程中出现的数据格式错误信息将被写入error_table_name指定的错误信息表中，可以在并行导入结束后查询此错误信息表，获取详细的错误信息。此参数只在设置了reject_limit参数时有效。

> **说明**
>
> 如果为了兼容postgres开源接口，此语法建议用LOG INTO代替。

取值范围：字符串，要符合标识符的命名规范。

- **LOG INTO error_table_name**

  数据导入过程中出现的数据格式错误信息将被写入error_table_name指定的错误信息表中，可以在并行导入结束后查询此错误信息表，获取详细的错误信息。

  > **说明**
  >
  > 若没有指定PER NODE REJECT LIMIT参数，则此参数不起作用。

  取值范围：字符串，要符合标识符的命名规范。

- **REMOTE LOG 'name'**

  数据导入过程中出现的数据格式错误信息将被写到GDS端以文件方式保存。name为错误数据文件的文件名前缀。

- **PER NODE REJECT LIMIT 'value'**

  指定本次数据导入过程中每个DN实例上允许出现的数据格式错误的数量，如果有一个DN实例上的错误数量大于设定值，本次导入失败，报错退出。

  须知

  此语法指定的是单个节点的错误容忍度。

  数据格式错误是指缺少或者多出字段值，数据类型错误或者编码错误。对于非数据格式错误，一旦发生就将导致整个数据扫描失败。

  取值范围：整型值，unlimited（无限），不指定该参数时，有错误信息立即返回。

- **TO { GROUP groupname | NODE (nodename [, ...]) }**

  TO GROUP目前不支持使用。TO NODE主要供内部扩容工具使用，一般用户不应使用。

示例

创建外表customer_ft，用来以TEXT格式导入GDS服务器10.10.123.234上的数据：

```
CREATE FOREIGN TABLE customer_ft
(
    c_customer_sk             integer               ,
    c_customer_id             char(16)              ,
    c_current_cdemo_sk        integer               ,
    c_current_hdemo_sk        integer               ,
    c_current_addr_sk         integer               ,
    c_first_shipto_date_sk    integer               ,
    c_first_sales_date_sk     integer               ,
    c_salutation              char(10)              ,
    c_first_name              char(20)              ,
    c_last_name               char(30)              ,
    c_preferred_cust_flag     char(1)               ,
    c_birth_day               integer               ,
    c_birth_month             integer               ,
    c_birth_year              integer               ,
    c_birth_country           varchar(20)           ,
    c_login                   char(13)              ,
    c_email_address           char(50)              ,
```

```
    c_last_review_date        char(10)
)
    SERVER gsmpp_server
    OPTIONS
(
    location 'gsfs://10.10.123.234:5000/customer1*.dat',
    FORMAT 'TEXT' ,
    DELIMITER '|',
    encoding 'utf8',
    mode 'Normal')
READ ONLY;
```

创建外表foreign_HR_staffS_ft，用来以TEXT格式导入GDS服务器192.168.0.90和192.168.0.91上的数据，导入过程错误信息将记录到err_HR_staffS中。本次数据导入允许出现的数据格式错误个数为2。

```
CREATE FOREIGN TABLE foreign_HR_staffS_ft
(
  staff_ID       NUMBER(6) ,
  FIRST_NAME     VARCHAR2(20),
  LAST_NAME      VARCHAR2(25),
  EMAIL          VARCHAR2(25),
  PHONE_NUMBER   VARCHAR2(20),
  HIRE_DATE      DATE,
  employment_ID  VARCHAR2(10),
  SALARY         NUMBER(8,2),
  COMMISSION_PCT NUMBER(2,2),
  MANAGER_ID     NUMBER(6),
  section_ID  NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'gsfs://192.168.0.90:5000/* | gsfs://192.168.0.91:5000/*', format
'TEXT', delimiter E'\x08',  null '',reject_limit '2') WITH err_HR_staffS_ft;
```

建立外表，用来以CSV格式导入input_data目录下存放在各个节点名文件下的所有文件。

```
CREATE FOREIGN TABLE foreign_HR_staffS_ft1
(
  staff_ID       NUMBER(6) ,
  FIRST_NAME     VARCHAR2(20),
  LAST_NAME      VARCHAR2(25),
  EMAIL          VARCHAR2(25),
  PHONE_NUMBER   VARCHAR2(20),
  HIRE_DATE      DATE,
  employment_ID  VARCHAR2(10),
  SALARY         NUMBER(8,2),
  COMMISSION_PCT NUMBER(2,2),
  MANAGER_ID     NUMBER(6),
  section_ID     NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'file:///input_data/*', format 'csv', quote E'\x08', mode 'private',
delimiter ',') WITH err_HR_staffS_ft1;
```

建立外表，用来以CSV格式导出数据到output_data目录下。

```
CREATE FOREIGN TABLE foreign_HR_staffS_ft2
(
  staff_ID       NUMBER(6) ,
  FIRST_NAME     VARCHAR2(20),
  LAST_NAME      VARCHAR2(25),
  EMAIL          VARCHAR2(25),
  PHONE_NUMBER   VARCHAR2(20),
  HIRE_DATE      DATE,
  employment_ID  VARCHAR2(10),
  SALARY         NUMBER(8,2),
  COMMISSION_PCT NUMBER(2,2),
  MANAGER_ID     NUMBER(6),
  section_ID  NUMBER(4)
) SERVER gsmpp_server OPTIONS (location 'file:///output_data/', format 'csv', quote E'\x08', delimiter '|',
header 'on') WRITE ONLY;
```


相关链接

ALTER FOREIGN TABLE (GDS导入导出)，DROP FOREIGN TABLE

12.35 CREATE FOREIGN TABLE (SQL on OBS or Hadoop)

功能描述

在当前数据库创建一个HDFS或OBS外表，用来访问存储在HDFS或者OBS分布式集群文件系统上的结构化数据。也可以导出ORC格式数据到HDFS或者OBS上。

数据存储在OBS：数据存储和计算分离，集群存储成本低，存储量不受限制，并且集群可以随时删除，但计算性能取决于OBS访问性能，相对HDFS有所下降，建议在数据计算不频繁场景下使用。

数据存储在HDFS：数据存储和计算不分离，集群成本较高，计算性能高，但存储量受磁盘空间限制，删除集群前需将数据导出保存，建议在数据计算频繁场景下使用。

说明

实时数仓（单机部署）暂不支持OBS和HDFS外表导入导出功能。

注意事项

- HDFS外表与OBS外表分为只读外表和只写外表，只读外表用于查询操作，只写外表可以将GaussDB(DWS)中的数据导出到分布式文件系统中。
- 此方式支持ORC、TEXT、CSV、CARBONDATA、PARQUET和JSON格式的导入查询，以及ORC、CSV和TEXT格式的导出。
- 该方式需要用户手动创建外部服务器，具体请参见CREATE SERVER。
- 若手动创建Server时指定foreign data wrapper为HDFS_FDW或者DFS_FDW，创建只读外表时需DISTRIBUTE BY子句指定分布方式。

语法格式

创建外表。

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
( [ { column_name type_name
   [ { [CONSTRAINT constraint_name] NULL |
   [CONSTRAINT constraint_name] NOT NULL |
     column_constraint [...]} ] |
     table_constraint [, ...]} [, ...] ] )
   SERVER server_name
   OPTIONS ( { option_name ' value ' } [, ...] )
   [ {WRITE ONLY | READ ONLY}]
   DISTRIBUTE BY {ROUNDROBIN | REPLICATION}

   [ PARTITION BY ( column_name ) [ AUTOMAPPED ] ] ;
```

- 其中column_constraint为：

  ```
  [CONSTRAINT constraint_name]
  {PRIMARY KEY | UNIQUE}
  [NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
  ```

- 其中table_constraint为：

```
[CONSTRAINT constraint_name]
{PRIMARY KEY | UNIQUE} (column_name)
[NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
```

参数说明

- **IF NOT EXISTS**

  如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。

- **table_name**

  外表的表名。

  取值范围：字符串，要符合标识符的命名规范。

- **column_name**

  外表中的字段名。可以选择多个字段名，中间用“,”隔开。

  取值范围：字符串，要符合标识符的命名规范。

  说明

  JSON对象由嵌套或并列的name-value对组成，具有顺序无关性，当导入JSON格式数据时，需要通过字段名与name的自动对应来确定字段与value的对应关系。用户需要定义恰当的字段名，否则可能导致导入结果不符合预期。字段名与name的自动对应规则如下：

  - 无嵌套无数组的情况下，字段名应当与name一致，不区分大小写。
  - 字段名使用‘_’字符拼接两个name，标识嵌套关系。
  - 字段名使用‘#’字符加十进制非负整数‘n’标识数组的第n个元素（从0开始）。

  例如，要导入JSON对象{"A" : "simple", "B" : {"C" : "nesting"}, "D" : ["array", 2, {"E" : "complicated"}]}中的每个元素，外表字段名应当分别定义为a、b、b_c、d、d#0、d#1、d#2、d#2_e，字段的定义顺序不会影响导入结果的正确性。

- **type_name**

  字段的数据类型。

- **constraint_name**

  外表的表约束名。

- **{ NULL | NOT NULL }**

  标识此列是否允许NULL值。

  在创建表时，对于列的约束NULL/NOT NULL，并不能保证该表在HDFS系统中的数据为NULL或者NOT NULL，数据的一致性由用户保证。所以需要由用户判断该列是否一定不为空或者一定为空，在建表的时候选用NULL或NOT NULL。(优化器对列为NULL/NOT NULL做了优化处理，会产生更优的计划。)

- **SERVER server_name**

  外表的server名字。允许用户自定义名字。

  取值范围：字符串，要符合标识符的命名规范，并且这个server必须存在。

- **OPTIONS ({ option_name ' value ' } [, ...])**

  用于指定外表数据的各类参数，参数类型如下所示。

  - header

    指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。header只能用于CSV格式的文件中。

    如果header选项为on，则数据文件第一行会被识别为标题行，导出时会忽略此行。如果header为off，而数据文件中第一行会被识别为数据。

取值范围：true/on，false/off。缺省值为false/off。

- quote

  CSV格式文件下的引号字符，缺省值为双引号。

  说明

  quote参数不能和分隔符、null参数相同。

  quote参数只能是单字节的字符。

  推荐不可见字符作为quote，例如0x07，0x08，0x1b等。

- escape

  CSV格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。

  缺省值为双引号。当与quote值相同时，会被替换为'\0'。

- location

  OBS外表参数，指定存储在OBS上的文件路径，多个桶的数据源数据之间使用分隔符‘|’进行分割，例如：LOCATION 'obs://bucket1/folder/ | obs://bucket2/'，数据库将会扫描指定路径文件夹下面的所有对象。

  当访问DLI多版本表时，无需指定location参数。

- format：外表中数据源文件的格式。

  - HDFS外表READ ONLY外表支持ORC、TEXT、JSON、CSV、PARQUET文件格式，而WRITE ONLY外表只支持ORC文件格式。
  - OBS外表READ ONLY外表支持ORC、TEXT、JSON、CSV、CARBONDATA、PARQUET文件格式，而WRITE ONLY外表只支持ORC文件格式。

  说明

  对于JSON格式数据，仅支持JSON对象（object，最外层由{}构造）导入，不支持JSON数组（array，最外层由[]构造）导入，但支持JSON对象内部数组的导入。

- foldername：外表中数据源文件目录，即表数据目录在HDFS文件系统和OBS上对应的文件目录。此选项对WRITE ONLY外表为必选项，对READ ONLY外表为可选项。

  当访问DLI多版本表时，无需指定foldername参数。

- encoding：外表中数据源文件的编码格式名称，缺省为utf8。此选项为可选参数。
- totalrows：可选参数，估计表的行数，仅OBS外表使用。由于OBS上文件可能很多，执行analyze可能会很慢，通过此参数让用户设置一个预估的值，使优化器能通过这个值做大小表的估计。一般预估值和实际值相近时，查询效率较高。
- filenames：外表中数据源文件，以","间隔。

说明

- 推荐通过使用foldername参数指定数据源的位置，对于只读外表filenames参数与foldername参数两者必有其一，而只写外表只能通过foldername指定。
- foldername为绝对目录时，前后必须有'/', 多个路径用', '分隔。
- 查询分区表时，会先根据分区信息进行剪枝，然后查询满足条件的数据文件。由于剪枝操作会涉及多次扫描HDFS分区目录内容，不建议使用重复度非常小的列作为分区列，因为这可能导致分区目录非常的多，增加对HDFS的查询压力。
- OBS只读外表不支持。

- delimiter

  指定数据文件行数据的字段分隔符，不指定则使用默认分隔符，TEXT格式的默认分隔符是水平制表符（tab）。

  说明

  - 分隔符不能是\r和\n。
  - 分隔符不能和null参数相同。
  - 分隔符不能包含“\”、“.”、数字和字母。
  - 数据文件中单行数据长度需<1GB，如果分隔符较长且数据列较多的情况下，会影响导出有效数据的长度。
  - 分隔符推荐使用多字符（例如'$^&'）和不可见字符（例如0x07、0x08、0x1b等）。
  - delimiter参数只在TEXT和CSV格式下有效。

  取值范围：

  支持多字符分隔符，但分隔符不能超过10个字节。

- eol

  指定导入数据文件换行符样式。

  取值范围：支持多字符换行符，但换行符不能超过10个字节。常见的换行符，如\r、\n、\r\n（设成0x0D、0x0A、0x0D0A效果是相同的），其他字符或字符串，如$、#。

  说明

  - eol参数只能用于TEXT格式的导入。
  - eol参数不能和分隔符、null参数相同。
  - eol参数不能包含：数字，字母和符号“.”。

- null

  用来指定数据文件中空值的表示。

  说明

  - null值不能是\r和\n，最大为100个字符。
  - null值不能是分隔符。
  - null参数只在TEXT和CSV格式下有效。

  取值范围：

  在TEXT格式下缺省值是\N。

- noescaping

  TEXT格式下，不对'\'和后面的字符进行转义。

说明

noescaping参数只在TEXT格式下有效。

取值范围：true/on，false/off。缺省值为false/off。

- fill_missing_fields

  当数据加载时，若数据源文件中一行的最后一个字段缺失时的处理方式。

  取值范围：true/on，false/off。缺省值为false/off。

  - 参数为true/on，当数据加载时，若数据源文件中一行数据的最后一个字段缺失，则把最后一个字段的值设置为NULL，不报错。
  - 参数为false/off，如果最后一个字段缺失会显示如下错误信息。

    ```
    missing data for column "tt"
    ```

  说明

  - TEXT格式下执行SELECT COUNT(*) 不会去解析具体字段，因此不会对字段缺失情况报错。
  - fill_missing_fields参数只在TEXT和CSV格式下有效。

- ignore_extra_data

  若数据源文件比外表定义列数多，是否会忽略多出的列。该参数只在数据导入过程中使用。

  取值范围：true/on，false/off。缺省值为false/off。

  - 参数为true/on，若数据源文件比外表定义列数多，则忽略行尾多出来的列。
  - 参数为false/off，若数据源文件比外表定义列数多，会显示如下错误信息。

    ```
    extra data after last expected column
    ```

须知

- 如果行尾换行符丢失，使两行变成一行时，设置此参数为true将导致后一行数据被忽略掉。
- TEXT格式下执行SELECT COUNT(*) 不会去解析具体字段，因此不会对多余的情况报错。
- ignore_extra_data参数只在TEXT和CSV格式下有效。

- date_format

  导入对于DATE类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法DATE格式。可参考**时间、日期处理函数和操作符**。

  说明

  - 对于指定为ORACLE兼容类型的数据库，则DATE类型内建为TIMESTAMP类型。在导入的时候，若需指定格式，可以参考下面的timestamp_format参数。
  - date_format参数只在TEXT和CSV格式下有效。

- time_format

  导入对于TIME类型指定格式。此语法仅对READ ONLY的外表有效。

取值范围：合法TIME格式，不支持时区。可参考**时间、日期处理函数和操作符**。

说明

time_format参数只在TEXT和CSV格式下有效。

- timestamp_format

  导入对于TIMESTAMP类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法TIMESTAMP格式，不支持时区。可参考**时间、日期处理函数和操作符**。

  说明

  timestamp_format参数只在TEXT和CSV格式下有效。

- smalldatetime_format

  导入对于SMALLDATETIME类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法SMALLDATETIME格式。可参考**时间、日期处理函数和操作符**。

  说明

  smalldatetime_format参数只在TEXT和CSV格式下有效。

- dataencoding

  在数据库编码与数据表的数据编码不一致时，该参数用于指定导出数据表的数据编码。比如数据库编码为Latin-1，而导出的数据表中的数据为UTF-8编码。此选项为可选项，如果不指定该选项，默认采用数据库编码。此语法仅对HDFS的WRITE ONLY外表有效。

  取值范围：该数据库编码支持转换的数据编码。

  说明

  dataencoding参数只对ORC格式的WRITE ONLY的HDFS外表有效。

- filesize

  指定WRITE ONLY外表的文件大小。此选项为可选项，不指定该选项默认分布式文件系统配置中文件大小的配置值。此语法仅对WRITE ONLY的外表有效。

  取值范围：[1, 1024]的整数。

  说明

  filesize参数只对ORC格式的WRITE ONLY的HDFS外表有效。

- compression

  指定ORC格式文件的压缩方式，此选项为可选项。此语法仅对WRITE ONLY的外表有效。

  取值范围：zlib，snappy，lz4。缺省值为snappy。

- version

  指定ORC格式的版本号，此选项为可选项。此语法仅对WRITE ONLY的外表有效。

  取值范围：目前仅支持0.12。缺省值为0.12。

- dli_project_id

  DLI服务对应的项目编号，可在管理控制台上获取项目ID，该参数仅支持server类型为DLI时设置。该参数仅8.1.1及以上版本支持。

- dli_database_name

  待访问的DLI多版本表所在的数据库名称，该参数仅支持server类型为DLI时设置。该参数仅8.1.1及以上版本支持。

- dli_table_name

  待访问的DLI多版本表的名称，该参数仅支持server类型为DLI时设置。该参数仅8.1.1及以上版本支持。

- checkencoding

  是否检查字符编码

  取值范围：low，high 。缺省值为low。

  说明

  TEXT格式下，导入非法字符容错规则如下：

  - 对于'\0'，容错后转换为空格；
  - 对于其他非法字符，容错后转换为问号；
  - 若checkencoding为low标识，导入时对于非法字符进行容错处理，则若NULL、DELIMITER设置为空格或问号则会通过如"illegal chars conversion may confuse null 0x20"等报错信息提示用户修改可能引起混淆的参数以避免导入错误。

  ORC格式下，导入非法字符容错规则如下：

  - checkencoding为low标识，若导入时检查到某个字段中包含非法字符，则自动将当前列当前行的字段整体替换为同样长度的‘？’字符；
  - checkencoding为high标识，若导入时检查到某个字段中包含非法字符，则报错退出。

- force_mapping

  JSON格式下，外表列无法匹配到正确的name-value键值对时的处理方式。

  取值范围：true，false。缺省值为true。

  - force_mapping为true，相应的列填null，该null与JSON定义中的null含义相同。
  - force_mapping为false，查询报错，提示不存在这样的列。

  说明

  由于对JSON对象没有限制，但外表字段定义需要符合GaussDB(DWS)的标识符规范（例如长度、字符等限制），因此这种导入方式可能导致异常：例如，字段无法正确标识JSON name、字段需重复定义等。建议使用容错性选项force_mapping或json操作符（可参考**JSON/JSONB函数和操作符**）来规避。

  JSON格式下执行SELECT COUNT(*) 不会去解析具体字段，因此不会对字段缺失、格式错误等情况报错。

表 12-22 text、csv、json、orc、carbondata、parquet 格式对 OBS 外表的 option 支持说明

参数名称	OBS					
-	TEXT	CSV	JSON	ORC	CARBONDATA	PARQUET

参数名称	OBS						
	READ ONLY	READ ONLY	READ ONLY	READ ONLY	WRITE ONLY	READ ONLY	READ ONLY
location	√	√	√	√	×	√	√
format	√	√	√	√	√	√	√
header	×	√	×	×	×	×	×
delimiter	√	√	×	×	×	×	×
quote	×	√	×	×	×	×	×
escape	×	√	×	×	×	×	×
null	√	√	×	×	×	×	×
noescaping	√	×	×	×	×	×	×
encoding	√	√	√	√	√	√	√
fill_missing_fields	√	√	×	×	×	×	×
ignore_extra_data	√	√	×	×	×	×	×
date_format	√	√	√	×	×	×	×
time_format	√	√	√	×	×	×	×
timestamp_format	√	√	√	×	×	×	×
smalldatetime_format	√	√	√	×	×	×	×
chunksize	√	√	√	×	×	×	×
filenames	×	×	×	×	×	×	×
foldername	√	√	√	√	√	√	√
dataencoding	×	×	×	×	×	×	×
filesize	×	×	×	×	×	×	×
compression	×	×	×	×	√	×	×
version	×	×	×	×	√	×	×
checkencoding	√	√	√	√	×	√	√

参数名称	OBS						
totalrows	√	√	√	√	×	×	×
force_mapping	×	×	√	×	×	×	×

表 12-23 text、csv、json、orc、parquet 格式对 HDFS 外表的 option 支持说明

参数名称	HDFS					
-	TEXT	CSV	JSON	ORC		PARQUET
	READ ONLY	READ ONLY	READ ONLY	READ ONLY	WRITE ONLY	READ ONLY
location	×	×	×	×	×	×
format	√	√	√	√	√	√
header	×	√	×	×	×	×
delimiter	√	√	×	×	×	×
quote	×	√	×	×	×	×
escape	×	√	×	×	×	×
null	√	√	×	×	×	×
noescaping	√	×	×	×	×	×
encoding	√	√	√	√	√	√
fill_missing_fields	√	√	×	×	×	×
ignore_extra_data	√	√	×	×	×	×
date_format	√	√	√	×	×	×
time_format	√	√	√	×	×	×
timestamp_format	√	√	√	×	×	×
smalldatetime_format	√	√	√	×	×	×
chunksize	√	√	√	×	×	×
filenames	√	√	√	√	×	√
foldername	√	√	√	√	√	√

参数名称	HDFS					
dataencoding	×	×	×	×	√	×
filesize	×	×	×	×	√	×
compression	×	×	×	×	√	×
version	×	×	×	×	√	×
checkencoding	√	√	√	√	√	√
totalrows	×	×	×	×	×	×
force_mapping	×	×	√	×	×	×

- WRITE ONLY | READ ONLY

  WRITE ONLY 指定创建HDFS/OBS的只写外表。

  READ ONLY 指定创建HDFS/OBS的只读外表。

  如果不指定创建的外表的类型，默认为只读外表。

- **DISTRIBUTE BY ROUNDROBIN**

  指定HDFS/OBS外表为ROUNDROBIN分布方式。

- **DISTRIBUTE BY REPLICATION**

  指定HDFS外表为REPLICATION分布方式。

- **PARTITION BY (column_name) AUTOMAPPED**

  column_name指定分区列。对于分区表，AUTOMAPPED表示HDFS分区外表指定的分区列会和HDFS数据中的分区目录信息自动对应，前提是必须保证HDFS分区外表指定分区列的顺序和HDFS数据中分区目录定义的顺序一致，该功能只适用于只读外表，只写外表不支持。

  说明

  - HDFS的只读和只写外表都支持分区表，但是只写外表只支持一级分区，不支持多级分区。
  - OBS只读外表支持分区表，OBS只写外表不支持分区表。
  - 不支持浮点类型和布尔类型的列作为分区列。
  - 分区字段长度限制可通过guc参数 "dfs_partition_directory_length" 调整。
  - 分区目录名称由 "分区列名=分区列值" 组成。此名称内如果包含特殊字符还会经过转义，所以转义前推荐长度不要超过 (dfs_partition_directory_length + 1) / 3 ，以此保证其转义后总长度不会因为超过 dfs_partition_directory_length 而出现报错。
  - 不推荐包含过长中文字符的列作为分区列。因为一个中文字符和一个英文字符所占的空间大小并不一致，用户不易计算最终的分区目录名称长度，更容易触发超过 dfs_partition_directory_length 长度限制的报错。

- **CONSTRAINT constraint_name**

  用于指定外表所建立的信息约束（Informational Constraint）的名字。

  取值范围：字符串，要符合标识符的命名规范。

- **PRIMARY KEY**

  主键约束，表示表里的一个或者一些字段只能包含唯一（不重复）的非NULL值。一个表只能声明一个主键。

- **UNIQUE**

  唯一约束，表示表里的一个或者多个字段的组合必须在全表范围内唯一。对于唯一约束，NULL被认为是互相不等的。

- **NOT ENFORCED**

  指定所建立的约束为信息约束，该约束不由数据库来保证，而由用户来保证。

- **ENFORCED**

  ENFORCED为默认值。预留参数，目前对于ENFORCED不支持。

- **PRIMARY KEY (column_name)**

  指定所建立的信息约束位于column_name列上。

  取值范围：字符串，要符合标识符的命名规范，并且这个column_name必须存在。

- **ENABLE QUERY OPTIMIZATION**

  利用信息约束对执行计划进行优化。

- **DISABLE QUERY OPTIMIZATION**

  禁止利用信息约束对执行计划优化。

信息约束（Informational Constraint）

在GaussDB(DWS)中，数据的约束完全由使用者保证，数据源数据能够严格遵守某种信息约束条件，能够加速对已经具有这种约束特征数据的查询。目前外表不支持索引，所以采取使用Informational Constraint信息优化Plan，提搞查询性能。

建立外表信息约束的约束条件：

- 只有用户保证表中的其中一列的非空值具有唯一性时才可以建立Informational Constraint，否则查询结果将与期望值不同。
- GaussDB(DWS)的Informational Constraint只支持PRIMARY KEY和UNIQUE两种约束。
- GaussDB(DWS)的Informational Constraint支持NOT ENFORCED属性，不支持ENFORCED属性。
- 一个表上的多列可以分别建立UNIQUE类型的Informational Constraint，但是PRIMARY KEY一个表中只能建立一个。
- 一个表的一列上可以建立多个Informational Constraint（由于一个列上有多个约束和一个的作用一致，所以不建议一个列上建立多个Informational Constraint），但是Primary Key类型只能建立一个。
- 不支持COMMENT。
- 不支持多列组合约束。
- ORC格式只写外表不支持同一个集群不同CN向同一外表并发导出。
- ORC格式只写外表的目录，只能用于GaussDB(DWS)的单个外表的导出目录，不能用于多个外表，并且其他组件不能向此目录写入其他文件。

示例 1

在HDFS通过HIVE导入TPC-H benchmark测试数据表part表及region表。part表的文件路径为/user/hive/warehouse/partition.db/part_4，region表的文件路径为/user/hive/warehouse/gauss.db/region_orc11_64stripe/。

1. 创建HDFS_Server，对应的foreign data wrapper为HDFS_FDW或者DFS_FDW。

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS (address
'10.10.0.100:25000,10.10.0.101:25000',hdfscfgpath '/opt/hadoop_client/HDFS/hadoop/etc/
hadoop',type'HDFS');
```

   说明

   - 在可选项options里面写入了HDFS集群对应的NameNode的IP地址及端口号。具体端口号请在MRS-HDFS服务配置中搜索参数“dfs.namenode.rpc.port”查看。本示例假设端口号为25000。
   - ‘10.10.0.100:25000,10.10.0.101:25000’中列出了两组NameNode的地址及端口号，分别表示HDFS的主NameNode及备NameNode，这里推荐使用该种主备方式填写。两组参量中间使用“,”进行分割。

2. 创建HDFS外表。表关联的HDFS server为hdfs_server，表ft_region对应的HDFS服务器上的文件格式为‘orc’，在HDFS文件系统上对应的文件目录为'/user/hive/warehouse/gauss.db/region_orc11_64stripe/'。
   - 创建不包含分区列的HDFS外表：

```
DROP FOREIGN TABLE IF EXISTS ft_region;
CREATE FOREIGN TABLE ft_region
(
    R_REGIONKEY INT4,
    R_NAME TEXT,
    R_COMMENT TEXT
)
SERVER
    hdfs_server
OPTIONS
(
    FORMAT 'orc',
    encoding 'utf8',
    FOLDERNAME '/user/hive/warehouse/gauss.db/region_orc11_64stripe/'
)
DISTRIBUTE BY
    roundrobin;
```

   - 创建包含分区列的HDFS外表：

```
CREATE FOREIGN TABLE ft_part
(
    p_partkey int,
    p_name text,
    p_mfgr text,
    p_brand text,
    p_type text,
    p_size int,
    p_container text,
    p_retailprice float8,
    p_comment text
)
SERVER
    hdfs_server
OPTIONS
(
    FORMAT 'orc',
    encoding 'utf8',
    FOLDERNAME '/user/hive/warehouse/partition.db/part_4'
)
DISTRIBUTE BY
    roundrobin
PARTITION BY
    (p_mfgr) AUTOMAPPED;
```


> **说明**
>
> GaussDB(DWS)支持2种文件指定方式：通过关键字filenames指定和通过foldername指定。推荐通过使用foldername进行指定。关键字distribute指定了表ft_region的存储分布方式。

3. 查看创建的外表：

```
SELECT * FROM pg_foreign_table WHERE ftrelid='ft_region'::regclass;
SELECT * FROM pg_foreign_table WHERE ftrelid='ft_part'::regclass;
```

示例 2

通过HDFS只写外表，将TPC-H benchmark测试数据表region中的数据导出至HDFS文件系统的/user/hive/warehouse/gauss.db/regin_orc/目录下。

1. 创建HDFS外表，对应的foreign data wrapper为HDFS_FDW或者DFS_FDW，同示例一。
2. 创建HDFS只写外表。

```
CREATE FOREIGN TABLE ft_wo_region
(
    R_REGIONKEY INT4,
    R_NAME TEXT,
    R_COMMENT TEXT
)
SERVER
    hdfs_server
OPTIONS
(
    FORMAT 'orc',
    encoding 'utf8',
    FOLDERNAME '/user/hive/warehouse/gauss.db/regin_orc/'
)
WRITE ONLY;
```

3. 通过只写外表向HDFS文件系统写入数据。

```
INSERT INTO ft_wo_region SELECT * FROM region;
```

示例 3

关于包含信息约束（Informational Constraint）HDFS外表的相关操作。

- 创建含有信息约束（Informational Constraint）的HDFS外表。

```
CREATE FOREIGN TABLE ft_region  (
 R_REGIONKEY  int,
 R_NAME TEXT,
 R_COMMENT TEXT
  , primary key (R_REGIONKEY) not enforced)
SERVER hdfs_server
OPTIONS(format 'orc',
    encoding 'utf8',
 foldername '/user/hive/warehouse/gauss.db/region_orc11_64stripe')
DISTRIBUTE BY roundrobin;
```

- 查看region表是否有信息约束索引：

```
SELECT relname,relhasindex FROM pg_class WHERE oid='ft_region'::regclass;
       relname         | relhasindex
-----------------------+-------------
       ft_region       | f
(1 row)

SELECT conname, contype, consoft, conopt, conindid, conkey FROM pg_constraint WHERE conname
='ft_region_pkey';
   conname     | contype | consoft | conopt | conindid | conkey
---------------+---------+---------+--------+----------+--------
ft_region_pkey | p       | t       | t      |        0 | {1}
(1 row)
```


- 删除信息约束：

```
ALTER FOREIGN TABLE ft_region DROP CONSTRAINT ft_region_pkey RESTRICT;

SELECT conname, contype, consoft, conindid, conkey FROM pg_constraint WHERE conname
='ft_region_pkey';
 conname | contype | consoft | conindid | conkey
---------+---------+---------+----------+--------
(0 rows)
```

- 添加一个唯一信息约束：

```
ALTER FOREIGN TABLE ft_region ADD CONSTRAINT constr_unique UNIQUE(R_REGIONKEY) NOT
ENFORCED;
```

  删除唯一信息约束：

```
ALTER FOREIGN TABLE ft_region DROP CONSTRAINT constr_unique RESTRICT;
SELECT conname, contype, consoft, conindid, conkey FROM pg_constraint WHERE conname
='constr_unique';
```

- 添加一个唯一信息约束：

```
ALTER FOREIGN TABLE ft_region ADD CONSTRAINT constr_unique UNIQUE(R_REGIONKEY) NOT
ENFORCED disable query optimization;
SELECT relname,relhasindex FROM pg_class WHERE oid='ft_region'::regclass;
```

  删除唯一信息约束：

```
ALTER FOREIGN TABLE ft_region DROP CONSTRAINT constr_unique CASCADE;
```

示例 4

通过外表读取OBS上的json数据。

1. OBS上有如下json文件，json对象中存在嵌套、数组，部分对象的某些字段缺失，部分对象name重复。

```
{"A" : "simple1", "B" : {"C" : "nesting1"}, "D" : ["array", 2, {"E" : "complicated"}]}
{"A" : "simple2", "D" : ["array", 2, {"E" : "complicated"}]}
{"A" : "simple3", "B" : {"C" : "nesting3"}, "D" : ["array", 2, {"E" : "complicated3"}]}
{"B" : {"C" : "nesting4"},"A" : "simple4",  "D" : ["array", 2, {"E" : "complicated4"}]}
{"A" : "simple5", "B" : {"C" : "nesting5"}, "D" : ["array", 2, {"E" : "complicated5"}]}
```

2. 创建obs_server，对应的foreign data wrapper为DFS_FDW。

```
CREATE SERVER obs_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (
  ADDRESS 'obs.xxx.xxx.com',
  ACCESS_KEY 'xxxxxxxxx',
  SECRET_ACCESS_KEY 'yyyyyyyyyyyyy',
  TYPE 'OBS'
);
```

   说明

   - ADDRESS是OBS的终端节点（Endpoint），请根据实际替换。也是使用region参数，通过指定regionCode在region_map文件中查找对应的域名。
   - ACCESS_KEY和SECRET_ACCESS_KEY 是云账号体系访问密钥。请根据实际替换。
   - 认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。
   - TYPE表示创建的Server为OBS Server。请保持OBS取值不变。

3. 创建OBS外表json_f，定义字段名，以d#2_e为例，从命名可以看出该字段是数组d的第二个元素里嵌套的e对象。表关联的OBS服务器为obs_server。foldername为外表中数据源文件目录，即表数据目录在OBS上对应的文件目录。

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
CREATE FOREIGN TABLE json_f (
  a VARCHAR(10),
  b_c TEXT,
  d#1 INTEGER,
  d#2_e VARCHAR(30)
)SERVER obs_server OPTIONS (
    foldername '/xxx/xxx/',
    format 'json',
    encoding 'utf8',
    force_mapping 'true'
)distribute by roundrobin;
```

4. 查询外表json_f。由于容错性参数force_mapping默认打开，json对象缺失的字段会填NULL；json对象name重复的以最后一次出现的name为准。

```
SELECT * FROM json_f;
   a    |   b_c    | d#1 |    d#2_e
--------+----------+-----+--------------
 simple1 | nesting1 |   2 | complicated1
 simple2 |          |   2 | complicated2
 simple3 | nesting3 |   2 | complicated3
 simple4 | nesting4 |   2 | complicated4
 repeat  | nesting5 |   2 | complicated5
(5 rows)
```

示例 5

通过外表读取DLI多版本外表。DLI多版本外表示例仅8.1.1及以上版本支持。

1. 创建dli_server，对应的foreign data wrapper为DFS_FDW。

```
CREATE SERVER dli_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (
  ADDRESS 'obs.xxx.xxx.com',
  ACCESS_KEY 'xxxxxxxxx',
  SECRET_ACCESS_KEY 'yyyyyyyyyyyyy',
  TYPE 'DLI',
  DLI_ADDRESS 'dli.xxx.xxx.com',
  DLI_ACCESS_KEY 'xxxxxxxxx',
  DLI_SECRET_ACCESS_KEY 'yyyyyyyyyyyyy'
);
```

说明

- ADDRESS是OBS的终端节点（Endpoint）。DLI_ADDRESS是DLI的终端节点（Endpoint），请根据实际替换。
- ACCESS_KEY和SECRET_ACCESS_KEY 是云账号体系访问OBS服务的密钥。请根据实际替换。
- DLI_ACCESS_KEY和DLI_SECRET_ACCESS_KEY是云账号体系访问DLI服务的密钥。请根据实际替换。
- 认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。
- TYPE表示创建的Server为DLI Server。请保持DLI取值不变。

2. 创建访问DLI多版本的OBS外表customer_address，不包含分区列，表关联的DLI服务器为dli_server。其中project_id为xxxxxxxxxxxxxxx，dli上的database_name为database123，需要访问的table_name为table456，根据实际替换。

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
CREATE FOREIGN TABLE customer_address
(
```

```
    ca_address_sk             integer               not null,
    ca_address_id             char(16)              not null,
    ca_street_number          char(10)                      ,
    ca_street_name            varchar(60)                   ,
    ca_street_type            char(15)                      ,
    ca_suite_number           char(10)                      ,
    ca_city                   varchar(60)                   ,
    ca_county                 varchar(30)                   ,
    ca_state                  char(2)                       ,
    ca_zip                    char(10)                      ,
    ca_country                varchar(20)                   ,
    ca_gmt_offset             decimal(36,33)                ,
    ca_location_type          char(20)
)
SERVER dli_server OPTIONS (
    FORMAT 'ORC',
    ENCODING 'utf8',
    DLI_PROJECT_ID 'xxxxxxxxxxxxxx',
    DLI_DATABASE_NAME 'database123',
    DLI_TABLE_NAME 'table456'
)
DISTRIBUTE BY roundrobin;
```

3. 通过外表查询DLI多版本表的数据。

```
SELECT COUNT(*) FROM customer_address;
 count
-------
    20
(1 row)
```

相关链接

ALTER FOREIGN TABLE (For HDFS or OBS)，**DROP FOREIGN TABLE**

12.36 CREATE FOREIGN TABLE (OBS 导入导出)

功能描述

在当前数据库创建一个外表，用于OBS数据并行导入导出。该方式使用的SERVER为数据库默认创建的gsmpp_server。

说明

实时数仓（单机部署）暂不支持OBS外表导入导出功能。

注意事项

- 这种方式仅支持TEXT和CSV格式，并且需要额外指定OBS连接信息。对于OBS上的ORC、Carbondata等格式数据，不适用这种方式，请参考**CREATE FOREIGN TABLE (SQL on OBS or Hadoop)**。
- 创建的外表分为只读外表（READ ONLY）和只写外表（WRITE ONLY），缺省为只读外表。数据导入集群时，请将外表设为READ ONLY；导出时，请设为WRITE ONLY。
- 外表由命令执行者所有；
- OBS外表不需要显式指定分布方式，默认支持ROUNDROBIN分布方式；
- 所创建外表只对**信息约束（Informational Constraint）**约束生效。
- OBS导入导出数据时，不支持中文路径。

表 12-24 OBS 外表支持读写格式说明

数据类型	自建Server		gsmpp_server	
-	READ ONLY	WRITE ONLY	READ ONLY	WRITE ONLY
ORC	√	√	×	×
PARQUET	√	×	×	×
CARBONDATA	√	×	×	×
TEXT	√	√	√	√
CSV	√	√	√	√
JSON	√	×	×	×

语法格式

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
( { column_name type_name [column_constraint ]
    | LIKE source_table | table_constraint [, ...]} [, ...] )
SERVER gsmpp_server
OPTIONS ( { option_name ' value ' } [, ...] )
[ { WRITE ONLY | READ ONLY }]
[ WITH error_table_name | LOG INTO error_table_name]
[PER NODE REJECT LIMIT 'value'] ;
```

- 其中column_constraint为：
  ```
  [CONSTRAINT constraint_name]
  {PRIMARY KEY | UNIQUE}
  [NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
  ```
- 其中table_constraint为：
  ```
  [CONSTRAINT constraint_name]
  {PRIMARY KEY | UNIQUE} (column_name)
  [NOT ENFORCED [ENABLE QUERY OPTIMIZATION | DISABLE QUERY OPTIMIZATION] | ENFORCED]
  ```

参数概览

创建外表语法提供了多个参数，常用参数分类如下。

- 必需参数
  - **table_name**
  - **column_name**
  - **type_name**
  - **SERVER gsmpp_server**
  - **access_key**
  - **secret_access_key**
- **OPTIONS参数**
  - 外表的数据源位置参数**location**
  - 数据格式参数
    - **format**

    - **header**（仅支持CSV格式）
    - **delimiter**
    - **quote**（仅支持CSV格式）
    - **escape**（仅支持CSV格式）
    - **null**
    - **noescaping**（仅支持TEXT格式）
    - **encoding**
    - **eol**
    - **bom（仅支持CSV格式）**
  - 容错性参数
    - **fill_missing_fields**
    - **ignore_extra_data**
    - **compatible_illegal_chars**
    - **PER NODE REJECT LIMIT 'val...**
    - **LOG INTO error_table_name**
    - **WITH error_table_name**

参数说明

- **IF NOT EXISTS**

  如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。
- **table_name**

  外表的表名。

  取值范围：字符串，要符合标识符的命名规范。
- **column_name**

  外表中的字段名。

  取值范围：字符串，要符合标识符的命名规范。
- **type_name**

  字段的数据类型。
- **SERVER gsmpp_server**

  外表的server名字。对于导入导出的OBS外表，其server是初始数据库默认创建的，即gsmpp_server。
- **OPTIONS ({ option_name ' value ' } [, ...])**

  用于指定外表数据的各类参数。
  - encrypt

数据传输过程中使用HTTPS，否则使用HTTP，默认off。

- access_key

  OBS访问协议对应的AK值（由用户从服务界面上用户信息里获取），创建外表时AK值会加密保存到数据库的元数据表中。

- secret_access_key：

  OBS访问协议对应的SK值（由用户从服务界面上用户信息里获取），创建外表时SK值会加密保存到数据库的元数据表中。

- chunksize

  在DN中每个OBS读取线程的缓存大小，可指定范围8~512，默认大小为64，单位为MB。

- location

  外表的数据源位置，目前支持URL描述。多个URL使用‘|’分割。

  说明

  - 对于只读外表（默认为只读）的URL末尾可以指定到对象路径的前缀或直接指定到对象全路径。指定方式为obs://bucket/prefix。（其中，prefix是指对象路径的前缀。）例如：obs://mybucket/tpch/nation/
  - 对于obs://bucket/prefix格式，若显式指定region参数时，域名信息将会读取指定的region参数；若region参数不指定，则读取defaultRegion的值，即安装集群时指定的region。
  - 对于可写外表，URL不需要指定文件名。外部表数据源位置只可指定一个，并且要预先创建好对应目录。
  - 对于只读外表不能使用多个相同的URL地址。
  - 向外表中插入数据需要指定location。
  - location参数中前缀gsobs、obs均支持，都识别为OBS的信息，若为gsobs时，其中包含obs url、bucket、prefix，若为obs时则表示bucket、prefix。

  在实际导入导出数据时，location参数使用建议如下：

  - 导入时“location”建议指定到具体文件名。如果仅指定到OBS桶或目录，则会导入其中的所有文本文件。当数据格式不正确时，则会报错。如果设置了容错，则容错表可能导入大量数据。
  - 支持OBS单桶多文件导入，根据文件名前缀进行匹配，匹配到的文件都会被导入。

    例如，有以下两个数据文件，只要在“location”中指定前缀mybucket/input_data/product_info就能识别并导入这两个文件。

    ```
    mybucket/input_data/product_info.0
    mybucket/input_data/product_info.1
    ```

  - 导入时如果指定到文件名，例如“1.csv”，那么在此文件的桶或目录存在此名称为前缀的其他文件，也会被导入。即“1.csv1”、“1.csv22”等等，都会被自动导入。
  - 导入时，“location”中如果使用obs方式，支持多个url，并且用‘|’分隔；如果使用gsobs方式，则不支持多个路径。
  - 导出时“location”默认按目录处理。如果仅指定到自定义名称的文件，则导出时会以该文件为名称创建目录，然后再生成导出文件。文件名由GaussDB(DWS)自动生成。
  - 导出时“location”只支持一个路径。

- region

  可选参数，region参数指定regionCode，regionCode为云上的region信息。

  若显式指定此参数，域名信息将会读取指定的region参数；若此参数不指定，则读取defaultRegion的值，即安装集群时指定的region。

  说明

  TEXT、CSV格式的OBS导入导出外表格式参数使用说明如下：

  - location参数必选，其中前缀gsobs、obs均支持，都识别为OBS的信息，若为gsobs时，其中包含obs url、bucket、prefix，若为obs时则表示bucket、prefix。
  - 多个桶的数据源数据之间使用分隔符‘|’进行分割，LOCATION 'obs://bucket1/folder/ | obs://bucket2/'，数据库将会扫描指定路径文件夹下面的所有对象。

- format

  外表中数据源文件的格式。

  取值范围：CSV、TEXT，缺省值为TEXT。GaussDB(DWS)只支持CSV和TEXT格式。

  - CSV（逗号分隔文件格式）：
    - 格式的文件，可以有效处理数据列中的换行符，但对一些特殊字符处理有欠缺。
    - 由记录组成，每条记录被分隔符分隔为字段，且每条记录都有同样的字段序列。
  - TEXT（文本格式）：
    - 由换行符区分每条记录，由分隔符区分每个字段。可以有效处理一些特殊字符，但无法正确处理数据列中的换行符。

- header

  指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。

  OBS导出数据时不支持该参数为ture，使用缺省值false，不需要设置，表示导出的数据文件第一行不是标题行（即表头）。

  在导入数据时，如果header选项为on，则数据文本第一行会被识别为标题行，会忽略此行。如果header为off，而数据文件中第一行会被识别为数据。

  取值范围：true/on，false/off。缺省值为false/off。

- delimiter

  指定数据文件行数据的字段分隔符，不指定则使用默认分隔符，TEXT格式的默认分隔符是水平制表符（tab），CSV格式的默认分隔符为“,”。

  说明

  - TEXT格式，分隔符不能是\r和\n。
  - 分隔符不能和null参数相同，CSV格式数据的分隔符不能和quote参数相同。
  - TEXT格式数据的分隔符不能包含：字母、数字以及特殊字符“\”和“.”。
  - 数据文件中单行数据长度需<1GB，如果分隔符较长且数据列较多的情况下，会影响导出有效数据的长度。
  - 分隔符推荐使用多字符（例如'$^&'）和不可见字符（例如0x07、0x08、0x1b等）。

  取值范围：

  支持多字符分隔符，但分隔符不能超过10个字节。

- quote

  CSV格式文件下的引号字符，缺省值为双引号。

  说明

  - quote参数不能和分隔符、null参数相同。
  - quote参数只能是单字节的字符。
  - 推荐不可见字符作为quote，例如0x07，0x08，0x1b等。

- escape

  CSV格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。

  缺省值为双引号。当与quote值相同时，会被替换为'\0'。

- null

  用来指定数据文件中空值的表示方式。

  说明

  - null的值不能是\r和\n，最大为100个字符。
  - null值不能和分隔符、quote参数相同。

  取值范围：

  - 在TEXT格式下缺省值是\N。
  - CSV格式下缺省值是一个没有引号的空字符串。

- noescaping

  TEXT格式下，开启后不对'\'和后面的字符进行转义。

  说明

  noescaping参数只在TEXT格式下有效。

  取值范围：true/on，false/off。缺省值为false/off。

- encoding

  指定数据文件的编码格式名称，即需要以何编码格式对数据文件进行解析和校验/输出文件为何种编码格式。缺省值为当前数据库的默认客户端编码格式，即client_encoding。

  导入外表此处强烈建议指定为文件的编码格式，或根据文件的字符集在导入前对client_encoding进行设置。否则可能会导致不必要的解析、校验错误以及其导致的导入报错回滚，甚至非法数据入库。导出外表同样希望指定此选项，以避免导出采用默认字符集设置时与预期不符。

  在创建外表时此选项未指定，会在客户端给出对应Warning信息。

  说明

  - 目前OBS导入外表不支持解析带有多种字符集编码格式混合的文件。
  - 目前OBS导出外表不支持写出带有多种字符集编码格式混合的文件。

- fill_missing_fields

  当数据导入时，若数据源文件中一行的最后一个字段缺失的处理方式。

  取值范围：true/on，false/off。缺省值为false/off。

  - 参数为true/on，当数据导入时，若数据源文件中一行数据的最后一个字段缺失，则把最后一个字段的值设置为NULL，不报错。

- 参数为false/off，如果最后一个字段缺失会显示如下错误信息。
  ```
  missing data for column "tt"
  ```

– ignore_extra_data

  数据源文件中的字段比外表定义列数多时，是否忽略多出的列。该参数只在数据导入过程中使用。

  取值范围：true/on，false/off。缺省值为false/off。

  - 参数为true/on，若数据源文件比外表定义列数多，则忽略行尾多出来的列。
  - 参数为false/off，若数据源文件比外表定义列数多，会显示如下错误信息。
    ```
    extra data after last expected column
    ```

须知

如果行尾换行符丢失，使两行变成一行时，设置此参数为true将导致后一行数据被忽略掉。

– reject_limit

  指定本次数据导入允许出现的数据格式错误个数，当导入过程中出现的数据格式错误未达到限定值时，本次数据导入可以成功。

须知

此语法建议用PER NODE REJECT LIMIT 'value'替代。

数据格式错误是指缺少或者多出字段值，数据类型错误或者编码错误。对于非数据格式错误，一旦发生就将导致整个数据导入失败。

  取值范围：整型值、unlimited（无限制）。

  不指定该参数时，有错误信息立即返回。

– eol

  指定导入导出数据文件换行符样式。

  取值范围：支持多字符换行符，但换行符不能超过10个字节。常见的换行符，如\r、\n、\r\n（设成0x0D、0x0A、0x0D0A效果是相同的），其他字符或字符串，如$、#。

  说明

  - eol参数只能用于TEXT格式的导入导出，不支持CSV格式。
  - eol参数不能和分隔符、null参数相同。
  - eol参数不能包含：数字，字母和符号“.”。

– date_format

  导入对于DATE类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法DATE格式。可参考**时间、日期处理函数和操作符**。

说明

对于指定为ORACLE兼容类型的数据库，则DATE类型内建为TIMESTAMP类型。在导入的时候，若需指定格式，可以参考下面的timestamp_format参数。

- time_format

  导入对于TIME类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法TIME格式，不支持时区。

- timestamp_format

  导入对于TIMESTAMP类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法TIMESTAMP格式，不支持时区。

- smalldatetime_format

  导入对于SMALLDATETIME类型指定格式。此语法仅对READ ONLY的外表有效。

  取值范围：合法SMALLDATETIME格式。

- compatible_illegal_chars

  导入非法字符容错参数。此语法仅对READ ONLY的外表有效。

  取值范围：true/on，false/off。缺省值为false/off。

  - 参数为true/on，则导入时遇到非法字符进行容错处理，非法字符转换后入库，不报错，不中断导入。
  - 参数为false/off，导入时遇到非法字符进行报错，中断导入。

须知

Windows平台下OBS若按照文本格式读取数据文件，遇到0x1A会作为EOF符号结束数据读入造成解析错误，这是Windows平台的实现约束。由于OBS不支持BINARY形式读取，可将相应数据文件交由Linux平台下的OBS读取。

说明

导入非法字符容错规则如下：

（1）对于'\0'，容错后转换为空格；

（2）对于其他非法字符，容错后转换为问号；

（3）若compatible_illegal_chars为true/on标识导入时对于非法字符进行容错处理，则若NULL、DELIMITER、QUOTE、ESCAPE设置为空格或问号则会通过如"illegal chars conversion may confuse COPY escape 0x20"等报错信息提示用户修改可能引起混淆的参数以避免导入错误。

- bom

  标识CSV文件是否包含utf8 BOM字段。

  取值范围：true/on，false/off。

  缺省值：false

说明

仅在只读外表且文件编码为utf8时生效。

- **READ ONLY**

外表只读，该参数只供数据导入使用。

- **WRITE ONLY**

  外表只写。该参数只供数据导出使用。

- **WITH error_table_name**

  数据导入过程中出现的数据格式错误信息将被写入error_table_name指定的错误信息表中，可以在并行导入结束后查询此错误信息表，获取详细的错误信息。此参数只在设置了reject_limit参数时有效。

  说明

  如果为了兼容postgres开源接口，此语法建议用LOG INTO代替。该参数指定时错误表自动创建。

  取值范围：字符串，要符合标识符的命名规范。

- **LOG INTO error_table_name**

  数据导入过程中出现的数据格式错误信息将被写入error_table_name指定的错误信息表中，可以在并行导入结束后查询此错误信息表，获取详细的错误信息。

  说明

  - 若没有指定PER NODE REJECT LIMIT参数，则此参数不起作用。
  - 该参数指定时，错误表自动创建。

  取值范围：字符串，要符合标识符的命名规范。

- **PER NODE REJECT LIMIT 'value'**

  指定本次数据导入过程中每个DN实例上允许出现的数据格式错误的数量，如果有一个DN实例上的错误数量大于设定值，本次导入失败，报错退出。

  须知

  此语法指定的是单个节点的错误容忍度。

  数据格式错误是指缺少或者多出字段值，数据类型错误或者编码错误。对于非数据格式错误，一旦发生就将导致整个数据扫描失败。

  取值范围：整型值，unlimited（无限），不指定该参数时，有错误信息立即返回。

- **NOT ENFORCED**

  指定所建立的约束为信息约束，该约束不由数据库来保证，而由用户来保证。

- **ENFORCED**

  ENFORCED为默认值。预留参数，目前对于ENFORCED不支持。

- **PRIMARY KEY (column_name)**

  指定所建立的信息约束位于column_name列上。

  取值范围：字符串，要符合标识符的命名规范，并且这个column_name必须存在。

- **ENABLE QUERY OPTIMIZATION**

  利用信息约束对查询计划进行优化。

- **DISABLE QUERY OPTIMIZATION**

  禁止利用信息约束对查询计划优化。

示例

创建外表OBS_ft，用来以txt格式导入OBS上指定的对象数据到row_tbl表中：

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
DROP FOREIGN TABLE IF EXISTS OBS_ft;
NOTICE:  foreign table "obs_ft" does not exist, skipping
DROP FOREIGN TABLE

CREATE FOREIGN TABLE OBS_ft( a int, b int)SERVER gsmpp_server OPTIONS (location 'obs://gaussdbcheck/
obs_ddl/test_case_data/txt_obs_informatonal_test001',format 'text',encoding 'utf8',chunksize '32', encrypt
'on',ACCESS_KEY 'access_key_value_to_be_replaced',SECRET_ACCESS_KEY
'secret_access_key_value_to_be_replaced',delimiter E'\x08') read only;
CREATE FOREIGN TABLE

DROP TABLE row_tbl;
DROP TABLE

CREATE TABLE row_tbl( a int, b int);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

INSERT INTO row_tbl SELECT * FROM OBS_ft;
INSERT 0 3
```

相关链接

ALTER FOREIGN TABLE (For HDFS or OBS)，DROP FOREIGN TABLE

12.37 CREATE FOREIGN TABLE (SQL on other GaussDB(DWS))

功能描述

在当前数据库创建一个协同分析的外表，用来访问存储在协同分析其他集群数据库中的表。

该外表是只读的，只能用于查询操作，可直接使用SELECT语句查询其数据。

语法格式

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
( [ { column_name type_name | LIKE source_table } [, ...] ] )
SERVER server_name
OPTIONS ( { option_name ' value ' } [, ...] )
[ READ ONLY ]
[ DISTRIBUTE BY {ROUNDROBIN} ]
[ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ];
```

参数说明

- **IF NOT EXISTS**

如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。

- **table_name**

  外表的表名。

  取值范围：字符串，要符合标识符的命名规范。

- **column_name**

  外表中的字段名。可以选择多个字段名，中间用“,”隔开。

  取值范围：字符串，要符合标识符的命名规范。

  说明

  不允许在列上创建约束，也不允许创建索引。

- **type_name**

  字段的数据类型。

  说明

  不允许使用序列当作类型和使用自定义类型。

- **SERVER** server_name

  server名称。允许用户自定义名称。

  取值范围：字符串，要符合标识符的命名规范，并且该server必须存在。

- **OPTIONS ({ option_name ' value ' } [, ...])**

  用于指定外表数据的各类参数，参数类型如下所示。

  - table_name：对应关联集群的表名，如果省略此option则认为该值和外表名字一样。
  - schema_name：对应关联集群的schema，如果省略此option则认为该值和外表所在的schema一样。
  - encoding：对应关联集群的编码，如果省略此option则使用关联集群的数据库编码。

- **READ ONLY**

  显示表是外表为只读。

- **DISTRIBUTE BY ROUNDROBIN**

  显示指定外表为ROUNDROBIN分布方式。

- **TO { GROUP groupname | NODE (nodename [, ...]) }**

  TO GROUP目前不支持使用。TO NODE主要供内部扩容工具使用，一般用户不应使用。

示例

1. 创建名称为server_remote的外部服务器，对应的foreign data wrapper为GC_FDW。

```
CREATE SERVER server_remote FOREIGN DATA WRAPPER GC_FDW OPTIONS (address '10.10.0.100:25000,10.10.0.101:25000',dbname 'test', username 'test', password '{Password}');
```

   说明

   在可选项options里面写入了关联集群CN对应的IP地址及端口号。这里推荐使用填写一个LVS的地址或者多个CN的地址。

2. 创建名称为region的外表：

```
DROP FOREIGN TABLE IF EXISTS region;
CREATE FOREIGN TABLE region
(
    R_REGIONKEY INT4,
    R_NAME TEXT,
    R_COMMENT TEXT
)
SERVER
    server_remote
OPTIONS
(
    schema_name 'test',
    table_name 'region',
    encoding 'gbk'
);
```

3. 查看创建的外表region：

```
\d+ region
```

```
gaussdb=> \d+ region
                          Foreign table "public.region"
   Column    |  Type   | Modifiers | FDW Options | Storage  | Stats target | Description
-------------+---------+-----------+-------------+----------+--------------+-------------
 r_regionkey | integer |           |             | plain    |              |
 r_name      | text    |           |             | extended |              |
 r_comment   | text    |           |             | extended |              |
Server: server_remote
FDW Options: (schema_name 'test', table_name 'region', encoding 'gbk')
FDW permition: read only
Has OIDs: no
Distribute By: ROUND ROBIN
Location Nodes: ALL DATANODES
```

相关链接

DROP FOREIGN TABLE，**ALTER FOREIGN TABLE (SQL on other GaussDB(DWS))**

12.38 CREATE FUNCTION

功能描述

创建一个函数。

注意事项

- 如果创建函数时参数或返回值带有精度，不进行精度检测。
- 创建函数时，函数定义中对表对象的操作建议都显式指定模式，否则可能会导致函数执行异常。
- 在创建函数时，函数内部通过SET语句设置current_schema和search_path无效。执行完函数search_path和current_schema与执行函数前的search_path和current_schema保持一致。
- 如果函数参数中带有出参，SELECT调用函数必须缺省出参，CALL调用函数适配Oracle必须指定出参，对于调用重载的带有PACKAGE属性的函数，CALL调用函数可以缺省出参，具体信息参见**CALL**的示例。
- 兼容PostgreSQL风格的函数或者带有PACKAGE属性的函数支持重载。在指定REPLACE的时候，如果参数个数、类型、返回值有变化，不会替换原有函数，而是会建立新的函数。
- SELECT调用可以指定不同参数来进行同名函数调用。由于语法CALL适配自Oracle，因此不支持调用不带有PACKAGE属性的同名函数。

- 在创建function时，不能在avg函数外面嵌套其他agg函数，或者其他系统函数。
- 在非逻辑集群模式下，暂不支持将返回值、参数以及变量设置为建在非系统默认安装Node Group的表，sql function内部语句暂不支持对建在非系统默认安装Node Group的表操作。
- 在逻辑集群模式下，如果函数返回值和参数是用户表类型，所有涉及表都必须在同一逻辑集群内；如果函数内部涉及对多个逻辑集群表操作，函数定义时不能为IMMUTABLE和SHIPPABLE类型，以避免函数被下推到DN执行。
- 在逻辑集群模式下，函数参数、返回值不能用%type引用表字段类型，否则会导致函数创建失败。
- 新创建的函数默认会给PUBLIC授予执行权限（详见**GRANT**）。用户可以选择收回PUBLIC默认执行权限，然后根据需要将执行权限授予其他用户，为了避免出现新函数能被所有人访问的时间窗口，应在一个事务中创建函数并且设置函数执行权限。

语法格式

- 兼容PostgreSQL风格的创建自定义函数语法。

```
CREATE [ OR REPLACE ] FUNCTION function_name
    ( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ]} [, ...] ] )
    [ RETURNS rettype [ DETERMINISTIC ] | RETURNS TABLE ( { column_name column_type } 
[, ...] )]
    LANGUAGE lang_name
    [
      {IMMUTABLE | STABLE | VOLATILE }
       | {SHIPPABLE | NOT SHIPPABLE}
       | WINDOW
       | [ NOT ] LEAKPROOF
       | {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
       | {[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER | AUTHID DEFINER | 
AUTHID CURRENT_USER}
       | {FENCED | NOT FENCED}
       | {PACKAGE}

       | COST execution_cost
       | ROWS result_rows
       | SET configuration_parameter { {TO | =} value | FROM CURRENT }}
    ][...]
    {
       AS 'definition'
       | AS 'obj_file', 'link_symbol'
    }
```

- Oracle风格的创建自定义函数的语法。

```
CREATE [ OR REPLACE ] FUNCTION function_name
    ( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ] } [, ...] ] )
    RETURN rettype [ DETERMINISTIC ]
    [
       {IMMUTABLE | STABLE | VOLATILE }
       | {SHIPPABLE | NOT SHIPPABLE}
       | {PACKAGE}
       | {FENCED | NOT FENCED}
       | [ NOT ] LEAKPROOF
       | {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
       | {[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER | 
AUTHID DEFINER | AUTHID CURRENT_USER
}
       | COST execution_cost
       | ROWS result_rows
       | SET configuration_parameter { {TO | =} value | FROM CURRENT

    ][...]

    {
```

```
    IS | AS
} plsql_body
/
```

参数说明

- **OR REPLACE**

  如果函数已存在，则重新定义。

- **function_name**

  要创建的函数名字（可以用模式修饰）。

  取值范围：字符串，要符合标识符的命名规范。

  说明

  如果创建的函数名与系统函数同名，建议指定schema。调用自定义函数时需指定schema，否则系统会优先调用系统函数。

- **argname**

  函数参数的名字。

  取值范围：字符串，要符合标识符的命名规范。

- **argmode**

  函数参数的模式。

  取值范围：IN，OUT，INOUT或VARIADIC。缺省值是IN。只有OUT模式的参数后面能跟VARIADIC。并且OUT和INOUT模式的参数不能用在RETURNS TABLE的函数定义中。

  说明

  VARIADIC用于声明数组类型的参数。

- **argtype**

  函数参数的类型。

- **expression**

  函数参数的默认表达式。

- **rettype**

  函数返回值的数据类型。

  如果存在OUT或IN OUT参数，可以省略RETURNS子句。如果存在，该子句必须和输出参数所表示的结果类型一致：如果有多个输出参数，则为RECORD，否则与单个输出参数的类型相同。

  SETOF修饰词表示该函数将返回一个集合，而不是单独一项。

- **DETERMINISTIC**

  为适配Oracle SQL语法，未实现功能，不推荐使用。

- **column_name**

  字段名称。

- **column_type**

  字段类型。

- **definition**

  一个定义函数的字符串常量，含义取决于语言。它可以是一个内部函数名字、一个指向某个目标文件的路径、一个SQL查询、一个过程语言文本。

- **LANGUAGE lang_name**

  用以实现函数的语言的名字。可以是SQL，internal，或者是用户定义的过程语言名字。为了保证向下兼容，该名字可以用单引号（包围）。若采用单引号，则引号内必须为大写。

- **WINDOW**

  表示该函数是窗口函数，替换函数定义时不能改变WINDOW属性。

  须知

  自定义窗口函数只支持LANGUAGE是internal，并且引用的内部函数必须是窗口函数。

- **IMMUTABLE**

  表示该函数在给出同样的参数值时总是返回同样的结果。

  如果函数的入参是常量，会在优化器阶段计算该函数的值。益处是可以尽早获取表达式的值，从而能更准确的进行代价估算，生成的执行计划也更优。

  用户自定义的IMMUTABLE的函数是会被自动下推到DN执行的，但是这样可能有潜在的风险，即如果用户错误定义了函数的IMMUTABLE属性，但是函数执行的过程并不是IMMUTABLE的，那么可能会导致结果错误等严重问题。因此，用户在指定函数的属性为IMMUTABLE的时候，要特别慎重。

  举例如下：

  a. 如果自定义函数中引用了表，视图等对象，那么该函数就不能定义为IMMUTABLE，因为当表的数据发生变化的时候，函数的返回值可能发生变化。
  b. 如果自定义函数中引用了STABLE/VOALATILE类型的函数，那么该函数不能定义为IMMUTABLE。
  c. 如果自定义函数中有不下推的因素，则该函数不能定义成IMMUTABLE，因为IMMUTABLE意味着要下推到DN执行，与函数内部的不下推因素相互冲突。典型场景例如，包含不下推的函数、语法等。
  d. 如果自定义函数中含有聚合运算，但聚合运算的运算需要生成STREAM计划才能完成计算的（部分结果在DN计算，部分结果在CN计算，例如listagg函数等）。

  同时，为了防止这种情况下可能出现严重问题，数据库内部可以通过设置behavior_compat_options=‘check_function_conflicts’来开启对函数定义冲突的检查，目前可以识别出上述**a**和**b**场景。

- **STABLE**

  表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同SQL语句之间变化。

- **VOLATILE**

  表示该函数值可以在一次表扫描内改变，因此不会做任何优化。

- **SHIPPABLE**

  NOT SHIPPABLE

  表示该函数是否可以下推到DN上执行。

  – 对于IMMUTABLE类型的函数，函数始终可以下推到DN上执行。

- 对于STABLE/VOLATILE类型的函数，仅当函数的属性是SHIPPABLE的时候，函数可以下推到DN执行。

  用户在定义函数的SHIPPABLE属性时也需特别慎重，SHIPPABLE意味着整个函数会下推到DN上执行，如果设置不当，会导致结果错误等严重问题。

  与定义IMMUTABLE属性一样，SHIPPABLE属性的定义也有诸多约束，简单来说就是函数内部不能有不可下推的因素，函数下推到单DN执行后，函数内部的计算逻辑仅依赖当前DN的数据集合。

  举例如下：

  i. 如果函数内部引用了表，并且表为HASH分布，那么该函数通常不能定义为SHIPPABLE。
  ii. 函数内部有不可下推的因素，函数，语法等，那么该函数不能定义为SHIPPABLE，可参考语句下推调优。
  iii. 函数内部的计算过程可能需要跨DN数据，这种情况该函数通常不能定义为SHIPPABLE，例如一些聚合运算等。

- **PACKAGE**

  表示该函数是否支持重载。PostgreSQL风格的函数本身就支持重载，此参数主要是针对Oracle风格的函数。

  - 不允许package函数和非package函数重载或者替换。
  - package函数不支持VARIADIC类型的参数。
  - 不允许修改函数的package属性。

- **LEAKPROOF**

  指出该函数的参数只包括返回值。LEAKPROOF只能由系统管理员设置。

- **CALLED ON NULL INPUT**

  表明该函数的某些参数是NULL的时候可以按照正常的方式调用。该参数可以省略。

- **RETURNS NULL ON NULL INPUT**

  STRICT

  STRICT用于指定如果函数的某个参数是NULL，此函数总是返回NULL。如果声明了这个参数，当有NULL值参数时该函数不会被执行；而只是自动返回一个NULL结果。

  RETURNS NULL ON NULL INPUT和STRICT的功能相同。

- **EXTERNAL**

  目的是和SQL兼容，是可选的，这个特性适合于所有函数，而不仅是外部函数。

- **SECURITY INVOKER**

  AUTHID CURRENT_USER

  表明该函数将带着调用它的用户的权限执行。该参数可以省略。

  SECURITY INVOKER和AUTHID CURRENT_USER的功能相同。

- **SECURITY DEFINER**

  AUTHID DEFINER

  声明该函数将以创建它的用户的权限执行。

  AUTHID DEFINER和SECURITY DEFINER的功能相同。

- **FENCED**

  NOT FENCED

该函数只对用户定义的C函数生效，声明函数是在保护模式还是非保护模式下执行。如果函数声明为NOT FENCED模式，则函数的执行在CN或者DN进程中进行。如果函数声明为FENCED模式，则函数在新fork的进程执行，这样函数的异常不会影响CN或者DN进程。

FENCED/NOT FENCED模式的选择：

- 正在开发或者调试的Function使用FENCED模式。开发测试完成，使用NOT FENCED模式执行，减少fork进程以及通信的开销。
- 复杂的操作系统操作，例：打开文件，信号处理，线程处理等操作，使用FENCED模式。否则可能影响GaussDB(DWS)数据库的执行。
- 默认值为FENCED。

- **COST execution_cost**

  用来估计函数的执行成本。

  execution_cost以cpu_operator_cost为单位。

  取值范围：正数

- **ROWS result_rows**

  估计函数返回的行数。用于函数返回的是一个集合。

  取值范围：正数，默认值是1000行。

- **configuration_parameter**
  - **value**

    把指定的数据库会话参数值设置为给定的值。如果value是DEFAULT或者RESET，则在新的会话中使用系统的缺省设置。OFF关闭设置。

    取值范围：字符串

    - DEFAULT
    - OFF
    - RESET

    指定默认值。
  - **from current**

    取当前会话中的值设置为configuration_parameter的值。

- **plsql_body**

  PL/SQL存储过程体。

须知

当在函数中创建用户时，日志中会记录密码的明文。因此不建议用户在函数中创建用户。

示例

创建定义为SQL查询的函数func_add_sql：

```
DROP FUNCTION IF EXISTS func_add_sql;
CREATE FUNCTION func_add_sql(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
```

```
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

创建利用参数名自增一个整数的函数func_increment_plsql：

```
DROP FUNCTION IF EXISTS func_increment_plsql;
CREATE OR REPLACE FUNCTION func_increment_plsql(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

创建返回RECORD类型的函数：

```
DROP FUNCTION IF EXISTS compute;
CREATE OR REPLACE FUNCTION compute(i int, out result_1 bigint, out result_2 bigint)
returns SETOF RECORD
as $$
begin
    result_1 = i + 1;
    result_2 = i * 10;
return next;
end;
$$language plpgsql;
```

创建记录返回包含多个输出参数的函数：

```
DROP FUNCTION IF EXISTS func_dup_sql;
CREATE FUNCTION func_dup_sql(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;
SELECT * FROM func_dup_sql(42);
```

创建计算两个整数的和的函数，并返回结果。若果输入为null，则返回null：

```
DROP FUNCTION IF EXISTS func_add_sql2;
CREATE FUNCTION func_add_sql2(num1 integer, num2 integer) RETURN integer
AS
BEGIN
RETURN num1 + num2;
END;
/
```

创建package属性的重载函数：

```
CREATE OR REPLACE FUNCTION package_func_overload(col int, col2  int)
return integer package
as
declare
    col_type text;
begin
     col := 122;
         dbms_output.put_line('two int parameters ' || col2);
         return 0;
end;
/

CREATE OR REPLACE FUNCTION package_func_overload(col int, col2 smallint)
return integer package
as
declare
    col_type text;
begin
     col := 122;
         dbms_output.put_line('two smallint parameters ' || col2);
         return 0;
end;
/
```


相关链接

ALTER FUNCTION，DROP FUNCTION

12.39 CREATE GROUP

功能描述

创建一个新用户组。

注意事项

CREATE GROUP是CREATE ROLE的别名，非SQL标准语法，不推荐使用，建议用户直接使用CREATE ROLE替代。

语法格式

```
CREATE GROUP group_name [ [ WITH ] option [ ... ] ]
    [ ENCRYPTED | UNENCRYPTED ] { PASSWORD | IDENTIFIED BY } { 'password' | DISABLE };
```

其中可选项action子句语法为：

```
where option can be:
{SYSADMIN | NOSYSADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | NODE GROUP logic_group_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
```

参数说明

请参考CREATE ROLE的参数说明。

相关链接

ALTER GROUP，DROP GROUP，CREATE ROLE

12.40 CREATE INDEX

功能描述

在指定的表上创建索引。

索引可以用来提高数据库查询性能，但是不恰当的使用将导致数据库性能下降。建议仅在匹配如下某条原则时创建索引：

- 经常执行查询的字段。
- 在连接条件上创建索引，对于存在多字段连接的查询，建议在这些字段上建立组合索引。例如，select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b，可以在t1表上的a，b字段上建立组合索引。
- where子句的过滤条件字段上（尤其是范围条件）。
- 在经常出现在order by、group by和distinct后的字段。
- 对于点查询场景，推荐建立btree索引。

在分区表上创建索引与在普通表上创建索引的语法不太一样，使用时请注意，如分区表上不支持并行创建索引、不支持创建部分索引、不支持NULL FIRST特性。

注意事项

- 索引自身也占用存储空间、消耗计算资源，创建过多的索引将对数据库性能造成负面影响（尤其影响数据导入的性能，建议在数据导入后再建索引）。因此，仅在必要时创建索引。
- 索引定义里的所有函数和操作符都必须是immutable类型的，即结果只能依赖于其输入参数，而不受任何外部的影响（如另外一个表的内容或者当前时间），该限制可以确保该索引的行为是定义良好的。在一个索引上或WHERE语句中使用用户定义函数时，请将其标记为immutable类型函数。
- 在分区表上创建唯一索引时，索引项中必须包含分布列和所有分区键。
- GaussDB(DWS)在分区表上创建索引时只支持本地（LOCAL）索引，不支持全局（GLOBAL）索引。
- 列存表和HDFS表支持B-tree索引，不支持创建表达式索引、部分索引。
- 列存表支持通过B-tree索引建立唯一索引。
- 列存表和HDFS表支持的PSORT索引不支持创建表达式索引、部分索引和唯一索引。
- 列存表支持的GIN索引支持创建表达式索引，但表达式不能包含空分词、空列和多列，不支持创建部分索引和唯一索引。
- roundrobin表不支持创建主键/唯一索引。
- 对表执行CREATE INDEX或REINDEX操作时会触发索引重建（索引重建过程中会先把数据转储到一个新的数据文件中，重建完成之后会删除原始文件），当表比较大时，重建会消耗较多的磁盘空间。当磁盘空间不足时，要谨慎对待大表CREATE INDEX或REINDEX操作，防止触发集群只读。

语法格式

- 在表上创建索引。

```
CREATE [ UNIQUE ] INDEX [ [ schema_name. ] index_name ] ON table_name [ USING method ]
    ({ { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
{ FIRST | LAST } ] }[, ...] )
    [ COMMENT 'text' ]
    [ WITH ( {storage_parameter = value} [, ... ] ) ]

    [ WHERE predicate ];
```

- 在分区表上创建索引。

```
CREATE [ UNIQUE ] INDEX [ [ schema_name. ] index_name ] ON table_name [ USING method ]
    ( {{ column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
LAST ] }[, ...] )
    [ COMMENT 'text' ]
    LOCAL [ ( { PARTITION index_partition_name  } [, ...] ) ]
    [ WITH ( { storage_parameter = value } [, ...] ) ]
    ;
```

参数说明

- **UNIQUE**

  创建唯一性索引，每次添加数据时检测表中是否有重复值。如果插入或更新的值会引起重复的记录时，将导致一个错误。

  目前只有行存表B-tree索引和列存表的B-tree索引支持唯一索引。

- **schema_name**

  要创建的索引所在的模式名。指定的模式名需与表所在的模式相同。

- **index_name**

  要创建的索引名，索引的模式与表相同。索引名不能与数据库中已有的表名重复。

  取值范围：字符串，要符合标识符的命名规范。

- **table_name**

  需要为其创建索引的表的名字，可以用模式修饰。

  取值范围：已存在的表名。

- **USING method**

  指定创建索引的方法。

  取值范围：

  - btree：B-tree索引使用一种类似于B+树的结构来存储数据的键值，通过这种结构能够快速的查找索引。btree适合支持比较查询以及查询范围。
  - gin：GIN索引是倒排索引，可以处理包含多个键的值（比如数组）。
  - gist：Gist索引适用于几何和地理等多维数据类型和集合数据类型。
  - Psort：Psort索引。针对列存表进行局部排序索引。

  行存表支持的索引类型：btree（行存表缺省值）、gin、gist。列存表支持的索引类型：Psort（列存表缺省值）、btree、gin。

- **column_name**

  表中需要创建索引的列的名字（字段名）。

  如果索引方式支持多字段索引，可以声明多个字段。最多可以声明32个字段。

- **expression**

  创建一个基于该表的一个或多个字段的表达式索引，通常必须写在圆括弧中。如果表达式有函数调用的形式，圆括弧可以省略。

  表达式索引可用于获取对基本数据的某种变形的快速访问。比如，一个在upper(col)上的函数索引将允许WHERE upper(col) = 'JIM'子句使用索引。

在创建表达式索引时，如果表达式中包含IS NULL子句，则这种索引是无效的。此时，建议用户尝试创建一个部分索引。

- **COLLATE collation**

  COLLATE子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。

- **opclass**

  操作符类的名字。对于索引的每一列可以指定一个操作符类，操作符类标识了索引那一列的使用的操作符。例如一个B-tree索引在一个四字节整数上可以使用int4_ops；这个操作符类包括四字节整数的比较函数。实际上对于列上的数据类型默认的操作符类是足够用的。操作符类主要用于一些有多种排序的数据。例如，用户想按照绝对值或者实数部分排序一个复数。能通过定义两个操作符类然后当建立索引时选择合适的类。

- **ASC**

  指定按升序排序（默认）。本选项仅行存支持。

- **DESC**

  指定按降序排序。本选项仅行存支持。

- **NULLS FIRST**

  指定空值在排序中排在非空值之前，当指定DESC排序时，本选项为默认的。

- **NULLS LAST**

  指定空值在排序中排在非空值之后，未指定DESC排序时，本选项为默认的。

- **COMMENT 'text'**

  指定索引的注释信息。

- **WITH ({storage_parameter = value} [, ...])**

  指定索引方法的存储参数。

  取值范围：

  只有GIN索引支持FASTUPDATE，GIN_PENDING_LIST_LIMIT参数。GIN和Psort之外的索引都支持FILLFACTOR参数。

  - FILLFACTOR

    一个索引的填充因子（fillfactor）是一个介于10和100之间的百分数。

    取值范围：10~100

  - FASTUPDATE

    GIN索引是否使用快速更新。

    取值范围：ON，OFF

    默认值：ON

  - GIN_PENDING_LIST_LIMIT

    当GIN索引启用fastupdate时，设置该索引pending list容量的最大值。

    取值范围：64~INT_MAX，单位KB。

    默认值：gin_pending_list_limit的默认取决于GUC中gin_pending_list_limit的值（默认为4MB）

- **WHERE predicate**

  创建一个部分索引。部分索引是一个只包含表的一部分记录的索引，通常是该表中比其他部分数据更有用的部分。例如，有一个表，表里包含已记账和未记账的定单，未记账的定单只占表的一小部分而且这部分是最常用的部分，此时就可以

通过只在未记账部分创建一个索引来改善性能。另外一个可能的用途是使用带有UNIQUE的WHERE强制一个表的某个子集的唯一性。

取值范围：predicate表达式只能引用表的字段，它可以使用所有字段，而不仅是被索引的字段。目前，子查询和聚集表达式不能出现在WHERE子句里。

- **PARTITION index_partition_name**

  索引分区的名称。

  取值范围：字符串，要符合标识符的命名规范。

示例

- 在表上创建索引。

  创建示例表tpcds.ship_mode_t1：

```
DROP TABLE IF EXISTS tpcds.ship_mode_t1;
CREATE TABLE tpcds.ship_mode_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
)
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK);
```

  在表tpcds.ship_mode_t1上的SM_SHIP_MODE_SK字段上创建唯一索引：

```
CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
```

  在表tpcds.ship_mode_t1上的SM_SHIP_MODE_SK字段上创建索引时添加索引的注释。

```
CREATE INDEX ds_ship_mode_t1_index_comment ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK)
COMMENT 'index';
```

  在表tpcds.ship_mode_t1上的SM_SHIP_MODE_SK字段上创建指定B-tree索引。

```
CREATE INDEX ds_ship_mode_t1_index4 ON tpcds.ship_mode_t1 USING btree(SM_SHIP_MODE_SK);
```

  在表tpcds.ship_mode_t1上的SM_SHIP_MODE_SK字段上创建指定GIN索引。

```
CREATE INDEX ship_mode_index ON tpcds.ship_mode_t1 USING gin(SM_SHIP_MODE_SK);
```

  在表tpcds.ship_mode_t1上SM_CODE字段上创建表达式索引。

```
CREATE INDEX ds_ship_mode_t1_index2 ON tpcds.ship_mode_t1(SUBSTR(SM_CODE,1 ,4));
```

  在表tpcds.ship_mode_t1上的SM_SHIP_MODE_SK字段上创建SM_SHIP_MODE_SK大于10的部分索引。

```
CREATE UNIQUE INDEX ds_ship_mode_t1_index3 ON tpcds.ship_mode_t1(SM_SHIP_MODE_SK)
WHERE SM_SHIP_MODE_SK>10;
```

- 在分区表上创建创建索引。

  创建示例表tpcds.customer_address_p1。

```
DROP TABLE IF EXISTS tpcds.customer_address_p1;
CREATE TABLE tpcds.customer_address_p1
(
    CA_ADDRESS_SK             INTEGER               NOT NULL,
    CA_ADDRESS_ID             CHAR(16)              NOT NULL,
    CA_STREET_NUMBER          CHAR(10)                      ,
    CA_STREET_NAME            VARCHAR(60)                   ,
    CA_STREET_TYPE            CHAR(15)                      ,
    CA_SUITE_NUMBER           CHAR(10)                      ,
    CA_CITY                   VARCHAR(60)                   ,
    CA_COUNTY                 VARCHAR(30)                   ,
    CA_STATE                  CHAR(2)                       ,
    CA_ZIP                    CHAR(10)                      ,
    CA_COUNTRY                VARCHAR(20)                   ,
```

```
    CA_GMT_OFFSET             DECIMAL(5,2)                ,
    CA_LOCATION_TYPE          CHAR(20)
)
DISTRIBUTE BY HASH(CA_ADDRESS_SK)
PARTITION BY RANGE(CA_ADDRESS_SK)
(
   PARTITION p1 VALUES LESS THAN (3000),
   PARTITION p2 VALUES LESS THAN (5000) ,
   PARTITION p3 VALUES LESS THAN (MAXVALUE)
)
ENABLE ROW MOVEMENT;
```

创建分区表索引ds_customer_address_p1_index1，不指定索引分区的名字。

```
CREATE INDEX ds_customer_address_p1_index1 ON tpcds.customer_address_p1(CA_ADDRESS_SK)
LOCAL;
```

创建分区表索引ds_customer_address_p1_index2，并指定索引分区的名字。

```
CREATE INDEX ds_customer_address_p1_index2 ON tpcds.customer_address_p1(CA_ADDRESS_SK)
LOCAL
(
    PARTITION CA_ADDRESS_SK_index1,
    PARTITION CA_ADDRESS_SK_index2,
    PARTITION CA_ADDRESS_SK_index3
)
;
```

创建分区表索引ds_customer_address_p1_index_comment，并添加索引注释。

```
CREATE INDEX ds_customer_address_p1_index_comment ON
tpcds.customer_address_p1(CA_ADDRESS_SK) COMMENT 'index' LOCAL
(
    PARTITION CA_ADDRESS_SK_index1,
    PARTITION CA_ADDRESS_SK_index2,
    PARTITION CA_ADDRESS_SK_index3
)
;
```

相关链接

ALTER INDEX，**DROP INDEX**

12.41 CREATE REDACTION POLICY

功能描述

对表创建数据脱敏策略。

注意事项

- 只有表对象的属主具有创建脱敏策略的权限。
- 仅支持在普通表创建数据脱敏策略，不支持为系统表、HDFS表、外表、临时表、UNLOGGED表以及视图和函数对象创建脱敏策略。
- 不支持通过同义词向普通表对象创建脱敏策略。
- 表对象与脱敏策略间一一对应。一个脱敏策略是表对象所有脱敏列的集合，可以给脱敏表对象的多个列字段指定脱敏函数，且不同脱敏列对象可以采用不同的脱敏函数。
- 创建脱敏策略时，默认策略生效，即属性enable值为true。
- 具有sysadmin权限的用户，可跳过脱敏策略检查，对脱敏列数据一直具有可见性，即脱敏策略不生效。

- 支持通过指定角色匹配脱敏策略。

语法格式

```
CREATE REDACTION POLICY policy_name ON table_name
   [ WHEN (when_expression) ]
   [ ADD COLUMN column_name WITH redaction_function_name ( [ argument [, ...] ] )] [, ... ];
```

参数说明

- **policy_name**

  脱敏策略名称。

- **table_name**

  应用脱敏策略的表名。

- **WHEN (when_expression)**

  WHEN子句指定一个生效表达式。仅当此表达式为真时，脱敏策略才可能生效。

  说明

  查询语句涉及脱敏表对象时，仅当脱敏策略的WHEN子句表达式为真时，查询对脱敏列数据才可能不可见，即脱敏策略生效。通常，采用WHEN子句来限定脱敏策略的生效用户范围，具有较严格的约束规格。

  WHEN子句的规格约束如下：

  1. 表达式可以是AND、OR连接的多个子表达式。
  2. 每个子表达式仅支持=、<>、!=、>=、>、<=、<七种运算符，左右值只能取常量值或者下列系统常量值之一：SESSION_USER、CURRENT_USER、USER、CURRENT_ROLE、CURRENT_SCHEMA系统常量或者SYS_CONTEXT系统函数。
  3. 每个子表达式可以是IN、NOT IN表达式，左值可以是上述2中所列系统常量值，右值数组的每个元素必须是常量值。
  4. 每个子表达式可以是**pg_has_role(user, role, privilege)**系统函数。
  5. 当脱敏策略永远成立时，即对所有用户（含表对象属主）均生效，建议使用表达式（1=1）创建脱敏策略。
  6. WHEN子句缺省时，脱敏策略默认不生效，需用户手动指定WHEN子句表达式。

- **column_name**

  脱敏策略应用的表对象的列名。

- **function_name**

  对脱敏列应用的脱敏函数。

- **arguments**

  脱敏函数的参数列表。

  - MASK_NONE，表示不进行任何脱敏处理。
  - MASK_FULL，表示全脱敏成固定值。
  - MASK_PARTIAL，表示按指定的字符类型，数值类型或时间类型进行部分脱敏处理。

  说明

  除系统提供MASK_NONE、MASK_FULL、MASK_PARTIAL三种内置脱敏函数，也支持使用C语言、PL/PGSQL语言创建的用户自定义脱敏函数，函数规格，请参考**数据脱敏函数**。

示例

对指定用户创建脱敏策略。

1. 创建用户alice和matu：
```
CREATE ROLE alice PASSWORD '{Password}';
CREATE ROLE matu PASSWORD '{Password}';
```
2. 用户alice创建表对象emp并插入数据：
```
CREATE TABLE emp(id int, name varchar(20), salary NUMERIC(10,2));
INSERT INTO emp VALUES(1, 'July', 1230.10), (2, 'David', 999.99);
```
3. 用户alice为表对象emp创建脱敏策略mask_emp，字段salary对用户matu不可见：
```
CREATE REDACTION POLICY mask_emp ON emp WHEN(current_user = 'matu') ADD COLUMN salary
WITH mask_full(salary);
```
4. 用户alice授予用户matu表emp的SELECT权限：
```
GRANT SELECT ON emp TO matu;
```
5. 切至用户matu：
```
SET ROLE matu PASSWORD '{Password}';
```
6. 查询表emp,字段salary数据已脱敏：
```
SELECT * FROM emp;
```

对角色创建脱敏策略。

1. 创建角色redact_role：
```
CREATE ROLE redact_role PASSWORD '{Password}';
```
2. 将用户matu，alice加入角色redact_role：
```
GRANT redact_role to matu,alice;
```
3. 管理员用户创建表对象emp1并插入数据：
```
CREATE TABLE emp1(id int, name varchar(20), salary NUMERIC(10,2));
INSERT INTO emp1 VALUES(3, 'Rose', 2230.20), (4, 'Jack', 899.88);
```
4. 管理员用户为表对象emp1创建脱敏策略mask_emp1，使字段salary对角色redact_role不可见。
```
CREATE REDACTION POLICY mask_emp1 ON emp1 WHEN(pg_has_role(current_user, 'redact_role',
'member')) ADD COLUMN salary WITH mask_full(salary);
```
   若不指定用户，默认为当前用户current_user：
```
CREATE REDACTION POLICY mask_emp1 ON emp1 WHEN (pg_has_role('redact_role', 'member'))
ADD COLUMN salary WITH mask_full(salary);
```
5. 管理员用户授予用户matu表emp1的SELECT权限：
```
GRANT SELECT ON emp1 TO matu;
```
6. 切换至用户matu：
```
SET ROLE matu PASSWORD '{Password}';
```
7. 查询表emp1，字段salary数据已脱敏。
```
SELECT * FROM emp1;
```

相关链接

ALTER REDACTION POLICY，**DROP REDACTION POLICY**

12.42 CREATE ROW LEVEL SECURITY POLICY

功能描述

对表创建行访问控制策略。

对表创建行访问控制策略时，需打开该表的行访问控制开关（ALTER TABLE ... ENABLE ROW LEVEL SECURITY）策略才能生效，否则不生效。

当前行访问控制会影响数据表的读取操作（SELECT、UPDATE、DELETE），暂不影响数据表的写入操作（INSERT、MERGE INTO）。表所有者或系统管理员可以在USING子句中创建表达式，并在客户端执行数据表读取操作时，数据库后台在查询重写阶段会将满足条件的表达式拼接并应用到执行计划中。针对数据表的每一条元组，当USING表达式返回TRUE时，元组对当前用户可见，当USING表达式返回FALSE或NULL时，元组对当前用户不可见。

行访问控制策略名称是针对表的，同一个数据表上不能有同名的行访问控制策略；对不同的数据表，可以有同名的行访问控制策略。

行访问控制策略可以应用到指定的操作（SELECT、UPDATE、DELETE、ALL），ALL表示会影响SELECT、UPDATE、DELETE三种操作；定义行访问控制策略时，若未指定受影响的相关操作，默认为ALL。

行访问控制策略可以应用到指定的用户（角色），也可应用到全部用户（PUBLIC）；定义行访问控制策略时，若未指定受影响的用户，默认为PUBLIC。

注意事项

- 支持对行存表、行存分区表、列存表、列存分区表、复制表、unlogged表、hash表定义行访问控制策略。
- 不支持HDFS表、外表、临时表定义行访问控制策略。
- 不支持对视图定义行访问控制策略。
- 同一张表上可以创建多个行访问控制策略，一张表最多创建100个行访问控制策略。
- 具有管理员权限的用户和初始运维用户(Ruby)不受行访问控制影响，可以查看表的全量数据。
- 通过SQL语句、视图、函数、存储过程查询包含行访问控制策略的表，都会受影响。
- 不支持对行访问控制策略依赖的列进行类型修改。例如，不支持如下修改：
  ```
  ALTER TABLE public.all_data ALTER COLUMN role TYPE text;
  ```

语法格式

```
CREATE [ ROW LEVEL SECURITY ] POLICY policy_name ON table_name
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC } [, ...] ]
    USING ( using_expression )
```

参数说明

- **policy_name**

  行访问控制策略名称，同一个数据表上行访问控制策略名称不能相同。

- **table_name**

  行访问控制策略的表名。

- **PERMISSIVE**

  指定行访问控制策略的类型为宽容性策略。对于一个给定的查询，将使用“OR”操作符将所有的宽容性策略组合。行访问控制策略的类型默认为宽容性策略。

- **RESTRICTIVE**

  指定行访问控制策略的类型为限制性策略。对于一个给定的查询，将使用“AND”操作符将所有的限制性策略组合。

  须知

  至少需要一条宽容性策略允许对记录的访问。如果只有限制性策略存在，则不能访问任何记录。当宽容性和限制性策略共存时，只有当记录能通过至少一条宽容性策略以及所有的限制性策略时，该记录才能访问。

- **command**

  当前行访问控制影响的SQL操作，可指定操作包括：ALL、SELECT、UPDATE、DELETE。当未指定时，ALL为默认值，涵盖SELECT、UPDATE、DELETE操作。

  当command为SELECT时，SELECT类操作受行访问控制的影响，只能查看到满足条件（using_expression返回值为TRUE）的元组数据，受影响的操作包括SELECT，UPDATE ... RETURNING，DELETE ... RETURNING。

  当command为UPDATE时，UPDATE类操作受行访问控制的影响，只能更新满足条件（using_expression返回值为TRUE）的元组数据，受影响的操作包括UPDATE, UPDATE ... RETURNING, SELECT ... FOR UPDATE/SHARE。

  当command为DELETE时，DELETE类操作受行访问控制的影响，只能删除满足条件（using_expression返回值为TRUE）的元组数据，受影响的操作包括DELETE, DELETE ... RETURNING。

  行访问控制策略与适配的SQL语法关系参见下表：

  表 12-25 ROW LEVEL SECURITY 策略与适配 SQL 语法关系

Command	SELECT/ALL policy	UPDATE/ALL policy	DELETE/ALL policy
SELECT	Existing row	No	No
SELECT FOR UPDATE/SHARE	Existing row	Existing row	No
UPDATE	No	Existing row	No
UPDATE RETURNING	Existing row	Existing row	No
DELETE	No	No	Existing row
DELETE RETURNING	Existing row	No	Existing row

- **role_name**

  行访问控制影响的数据库用户。

  当未指定时，PUBLIC为默认值，PUBLIC表示影响所有数据库用户，可以指定多个受影响的数据库用户。

须知

系统管理员不受行访问控制特性影响。

- **using_expression**

  行访问控制的表达式（返回boolean值）。

  条件表达式中不能包含AGG函数和窗口（WINDOW）函数。在查询重写阶段，如果数据表的行访问控制开关打开，满足条件的表达式会添加到计划树中。针对数据表的每条元组，会进行表达式计算，只有表达式返回值为TRUE时，行数据对用户才可见（SELECT、UPDATE、DELETE）；当表达式返回FALSE时，该元组对当前用户不可见，用户无法通过SELECT语句查看此元组，无法通过UPDATE语句更新此元组，无法通过DELETE语句删除此元组。

示例

创建用户alice：

```
CREATE ROLE alice PASSWORD '{Password}';
```

创建用户bob：

```
CREATE ROLE bob PASSWORD '{Password}';
```

创建数据表public.all_data：

```
CREATE TABLE public.all_data(id int, role varchar(100), data varchar(100));
```

向数据表插入数据：

```
INSERT INTO all_data VALUES(1, 'alice', 'alice data');
INSERT INTO all_data VALUES(2, 'bob', 'bob data');
INSERT INTO all_data VALUES(3, 'peter', 'peter data');
```

将表all_data的读取权限赋予alice和bob用户：

```
GRANT SELECT ON all_data TO alice, bob;
```

打开行访问控制策略开关：

```
ALTER TABLE all_data ENABLE ROW LEVEL SECURITY;
```

创建行访问控制策略，当前用户只能查看用户自身的数据：

```
CREATE ROW LEVEL SECURITY POLICY all_data_rls ON all_data USING(role = CURRENT_USER);
```

查看表all_data相关信息：

```
\d+ all_data
                          Table "public.all_data"
 Column |          Type          | Modifiers | Storage  | Stats target | Description
--------+------------------------+-----------+----------+--------------+-------------
 id     | integer                |           | plain    |              |
 role   | character varying(100) |           | extended |              |
 data   | character varying(100) |           | extended |              |
Row Level Security Policies:
    POLICY "all_data_rls"
      USING (((role)::name = "current_user"()))
Has OIDs: no
Distribute By: HASH(id)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no, enable_rowsecurity=true
```

当前用户执行SELECT操作：

```
SELECT * FROM all_data;
 id | role  |    data
----+-------+------------
  1 | alice | alice data
  2 | bob   | bob data
  3 | peter | peter data
(3 rows)
EXPLAIN(COSTS OFF) SELECT * FROM all_data;
         QUERY PLAN
----------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Seq Scan on all_data
(3 rows)
```

切换至alice用户：

```
set role alice password '{Password}';
```

执行SELECT操作：

```
SELECT * FROM all_data;
 id | role  |    data
----+-------+------------
  1 | alice | alice data
(1 row)

EXPLAIN(COSTS OFF) SELECT * FROM all_data;
                        QUERY PLAN
-----------------------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Seq Scan on all_data
         Filter: ((role)::name = 'alice'::name)
 Notice: This query is influenced by row level security feature
(5 rows)
```

相关链接

DROP ROW LEVEL SECURITY POLICY

12.43 CREATE PROCEDURE

功能描述

创建一个新的存储过程。

注意事项

- 如果创建存储过程时参数或返回值带有精度，不进行精度检测。
- 创建存储过程时，存储过程定义中对表对象的操作建议都显示指定模式，否则可能会导致存储过程执行异常。
- 在创建存储过程时，存储过程内部通过SET语句设置current_schema和search_path无效。执行完函数search_path和current_schema与执行函数前的search_path和current_schema保持一致。
- 如果存储过程参数中带有出参，SELECT调用存储过程必须缺省出参，CALL调用存储过程适配Oracle，调用非重载函数时必须指定出参，对于重载的package函数，out参数可以缺省，具体信息参见**CALL**的示例。
- 存储过程指定package属性时支持重载。

- 在创建procedure时，不能在avg函数外面嵌套其他agg函数，或者其他系统函数。

语法格式

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name
    [ ( {[ argmode ] [ argname ] argtype [ { DEFAULT | := | = } expression ]}[,...]) ]
    [
      { IMMUTABLE | STABLE | VOLATILE }
      | { SHIPPABLE | NOT SHIPPABLE }
      | {PACKAGE}
      | [ NOT ] LEAKPROOF
      | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
      | {[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER | AUTHID DEFINER | AUTHID
CURRENT_USER}
      | COST execution_cost
      | ROWS result_rows
      | SET configuration_parameter { [ TO | = ] value | FROM CURRENT }
    ][ ... ]
 { IS | AS }
plsql_body
/
```

参数说明

- **OR REPLACE**

  当存在同名的存储过程时，替换原来的定义。

- **procedure_name**

  创建的存储过程名字，可以带有模式名。

  取值范围：字符串，要符合标识符的命名规范。

- **argmode**

  参数的模式。

  须知

  VARIADIC用于声明数组类型的参数。

  取值范围： IN，OUT，INOUT或VARIADIC。缺省值是IN。只有OUT模式的参数后面能跟VARIADIC。并且OUT和INOUT模式的参数不能用在RETURNS TABLE的过程定义中。

- **argname**

  参数的名字。

  取值范围：字符串，要符合标识符的命名规范。

- **argtype**

  参数的数据类型。

  取值范围：可用的数据类型。

- **IMMUTABLE、STABLE等**

  行为约束可选项。各参数的功能与CREATE FUNCTION类似，详细说明见 **5.18.17.13-CREATE FUNCTION**

- **plsql_body**

  PL/SQL存储过程体。

须知

当在存储过程体中进行创建用户等涉及用户密码相关操作时，系统表及csv日志中会记录密码的明文。因此不建议用户在存储过程体中进行涉及用户密码的相关操作。

说明

argument_name和argmode的顺序没有严格要求，推荐按照argument_name、argmode、argument_type的顺序使用。

示例

创建一个存储过程：

```
CREATE OR REPLACE PROCEDURE prc_add
(
    param1    IN    INTEGER,
    param2    IN OUT  INTEGER
)
AS
BEGIN
   param2:= param1 + param2;
   dbms_output.put_line('result is: '||to_char(param2));
END;
/
```

调用此存储过程：

```
SELECT prc_add(2,3);
```

创建一个参数模式为VARIADIC的存储过程：

```
CREATE OR REPLACE PROCEDURE pro_variadic (param1 VARIADIC int4[], param2  OUT  TEXT)
AS
BEGIN
    param2:= param1::text;
END;
/
```

执行此存储过程：

```
SELECT pro_variadic(VARIADIC param1=> array[1,2,3,4]);
```

创建带有package属性的存储过程：

```
CREATE OR REPLACE PROCEDURE package_func_overload(col int, col2 out varchar)
package
as
declare
    col_type text;
begin
     col2 := '122';
         dbms_output.put_line('two varchar parameters ' || col2);
end;
/
```

相关链接

DROP PROCEDURE，**CALL**

12.44 CREATE RESOURCE POOL

功能描述

创建一个资源池，并指定此资源池相关联的控制组。

注意事项

只要用户对当前数据库有CREATE权限，就可以创建资源池。

语法格式

```
CREATE RESOURCE POOL pool_name
    [WITH ({MEM_PERCENT=pct | CONTROL_GROUP="group_name" | ACTIVE_STATEMENTS=stmt |
MAX_DOP = dop | MEMORY_LIMIT='memory_size' | io_limits=io_limits | io_priority='io_priority' |
nodegroup="nodegroupname" | is_foreign=boolean }[, ... ])];
```

参数说明

- **pool_name**

  资源池名称。

  资源池名称不能和当前数据库里已有的资源池重名。

  取值范围：字符串，要符合标识符的命名规范。

- **group_name**

  控制组名称。

  说明

  - 设置控制组名称时，语法可以使用双引号，也可以使用单引号。
  - group_name对大小写敏感。
  - 不指定group_name时，默认指定的字符串为“Medium”，代表指定DefaultClass控制组的 “Medium”Timeshare控制组。
  - 若数据库管理员指定自定义Class组下的Workload控制组，如control_group的字符串为：“class1:workload1”；代表此资源池指定到class1控制组下的workload1控制组。也可同时指定Workload控制组的层次，如control_group的字符串为：“class1:workload1:1”。
  - 若数据库用户指定Timeshare控制组代表的字符串，即“Rush”、“High”、“Medium”或“Low”其中一种，如control_group的字符串为“High”；代表资源池指定到DefaultClass控制组下的“High”Timeshare控制组。
  - 多租户场景下，组资源池关联的控制组为Class级别，业务资源池关联Workload控制组。且不允许在各种资源池间相互切换。

  取值范围：字符串，要符合说明中的规则，其指定已创建的控制组。

- **stmt**

  资源池语句执行的最大并发数量。

  取值范围：数值型，-1~INT_MAX。

- **dop**

  资源池简单语句执行的最大并发数量。

  取值范围：数值型，1~INT_MAX。

- **memory_size**
  资源池最大使用内存。
  取值范围：字符串，内容范围1KB~2047GB。
- **mem_percent**
  资源池可用内存占全部内存或者组用户内存使用的比例。
  在多租户场景下，组用户和业务用户的mem_percent范围1-100，默认为20。
  在普通场景下，普通用户的mem_percent范围为0-100，默认值为0。

  说明

  mem_percent和memory_limit同时指定时，只有mem_percent起作用。
- **io_limits**
  该参数8.1.2版本中已废弃，为兼容历史版本保留该参数。
- **io_priority**
  该参数8.1.2版本中已废弃，为兼容历史版本保留该参数。
- **nodegroup**
  在逻辑集群模式下，指定资源池所属的逻辑集群名称。必须是存在的逻辑集群。
  如果逻辑集群名称包含大写字符、特殊符号或以数字开头，SQL语句中对逻辑集群名称需要加双引号。
- **is_foreign**
  在逻辑集群模式下，指定当前资源池用于控制没有关联本逻辑集群的普通用户的资源。这里的逻辑集群是由资源池nodegroup字段指定的。

  说明

  - nodegroup必须是存在的逻辑集群，不能是elastic_group和安装的nodegroup(group_version1)。
  - 如果指定了is_foreign为true，则资源池不能再关联用户，即不允许通过CREATE USER ... RESOURCE POOL语句来将该资源池配置给用户。该资源池自动检查用户是否关联到资源池指定的逻辑集群，如果用户没有关联到该逻辑集群，则这些用户在逻辑集群所包含的DN上运行将受到该资源池的资源控制。

示例

本示例假定用户已预先成功创建控制组。

创建一个默认资源池，其控制组为“DefaultClass”组下属的“Medium”Timeshare Workload控制组：

```
CREATE RESOURCE POOL pool1;
```

创建一个资源池，其控制组指定为“DefaultClass”组下属的"High" Timeshare Workload控制组：

```
CREATE RESOURCE POOL pool2 WITH (CONTROL_GROUP="High");
```

创建一个资源池，其控制组指定为“class1”组下属的“Low”Timeshare Workload控制组：

```
CREATE RESOURCE POOL pool3 WITH (CONTROL_GROUP="class1:Low");
```

创建一个资源池，其控制组指定为“class1”组下属的“wg1”Workload控制组：

```
CREATE RESOURCE POOL pool4 WITH (CONTROL_GROUP="class1:wg1");
```


创建一个资源池，其控制组指定为“class1”组下属的“wg2”Workload控制组：

```
CREATE RESOURCE POOL pool5 WITH (CONTROL_GROUP="class1:wg2:3");
```

相关链接

ALTER RESOURCE POOL，**DROP RESOURCE POOL**

12.45 CREATE ROLE

功能描述

创建角色。

角色是拥有数据库对象和权限的实体。在不同的环境中角色可以认为是一个用户，一个组或者兼顾两者。

注意事项

- 在数据库中添加一个新角色，角色无登录权限。
- 创建角色的用户必须具备CREATE ROLE的权限或者是系统管理员。

语法格式

```
CREATE ROLE role_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ] { PASSWORD |
IDENTIFIED BY } { 'password' | DISABLE };
```

其中角色信息设置子句option语法为：

```
{SYSADMIN | NOSYSADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_cluster_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN rol e_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
    | AUTHINFO 'authinfo'
    | PASSWORD EXPIRATION period
```

参数说明

- **role_name**

  角色名称。

  取值范围：字符串，要符合标识符的命名规范且最多为63个字符。

- **password**

  登录密码。

  密码规则如下：

  - 密码默认不少于8个字符。
  - 不能与用户名及用户名倒序相同。
  - 至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。使用范围外的字符会收到告警，但依然允许创建。

  取值范围：字符串。

- **DISABLE**

  默认情况下，用户可以更改自己的密码，除非密码被禁用。要禁用用户的密码，请指定DISABLE。禁用某个用户的密码后，将从系统中删除该密码，此类用户只能通过外部认证来连接数据库，例如：IAM认证、kerberos或ldap认证。只有管理员才能启用或禁用密码。普通用户不能禁用初始用户的密码。要启用密码，请运行ALTER USER并指定密码。

- **ENCRYPTED | UNENCRYPTED**

  控制密码存储在系统表里的密码是否加密。如果没有指定，那么缺省的行为由配置参数password_encryption_type控制。按照产品安全要求，密码必须加密存储，所以UNENCRYPTED在GaussDB(DWS)中禁止使用。因为系统无法对指定的加密密码字符串进行解密，所以如果目前的密码字符串已经是用SHA256加密的格式，则会继续照此存放，而不管是否声明了ENCRYPTED或UNENCRYPTED。这样就允许在dump/restore的时候重新加载加密的密码。

- **SYSADMIN | NOSYSADMIN**

  决定一个新角色是否为“系统管理员”，具有SYSADMIN属性的角色拥有系统最高权限。

  缺省为NOSYSADMIN。

- **AUDITADMIN | NOAUDITADMIN**

  定义角色是否有审计管理属性。

  缺省为NOAUDITADMIN。

- **CREATEDB | NOCREATEDB**

  决定一个新角色是否能创建数据库。

  新角色没有创建数据库的权限。

  缺省为NOCREATEDB。

- **USEFT | NOUSEFT**

  决定一个新角色是否能操作外表，包括：新建外表、删除外表、修改外表、读写外表。

  新角色没有操作外表的权限。

  缺省为NOUSEFT。

- **CREATEROLE | NOCREATEROLE**

决定一个角色是否可以创建新角色（也就是执行CREATE ROLE和CREATE USER）。一个拥有CREATEROLE权限的角色也可以修改和删除其他角色。

缺省为NOCREATEROLE。

- **INHERIT | NOINHERIT**

  这些子句决定一个角色是否“继承”它所在组的角色的权限。不推荐使用。

- **LOGIN | NOLOGIN**

  具有LOGIN属性的角色才可以登录数据库。一个拥有LOGIN属性的角色可以认为是一个用户。

  缺省为NOLOGIN。

- **REPLICATION | NOREPLICATION**

  定义角色是否允许流复制或设置系统为备份模式。REPLICATION属性是特定的角色，仅用于复制。

  缺省为NOREPLICATION。

- **INDEPENDENT | NOINDEPENDENT**

  定义私有、独立的角色。具有INDEPENDENT属性的角色，管理员对其进行的控制、访问的权限被分离，具体规则如下：

  - 未经INDEPENDENT角色授权，管理员无权对其表对象进行增、删、查、改、复制、授权操作。
  - 未经INDEPENDENT角色授权，管理员无权修改INDEPENDENT角色的继承关系。
  - 管理员无权修改INDEPENDENT角色的表对象的属主。
  - 管理员无权去除INDEPENDENT角色的INDEPENDENT属性。
  - 管理员无权修改INDEPENDENT角色的数据库密码，INDEPENDENT角色需管理好自身密码，密码丢失无法重置。
  - 管理员属性用户不允许定义修改为INDEPENDENT属性。

- **VCADMIN | NOVCADMIN**

  定义逻辑集群管理员角色。具有逻辑集群管理员属性的角色，和普通用户相比，有如下额外权限：

  - 在所关联逻辑集群中创建、修改和删除资源池的权限。
  - 将所关联的逻辑集群的访问权限授予其他用户或角色，或回收其他用户或角色对关联逻辑集群的访问权限。

- **CONNECTION LIMIT**

  声明该角色在单个CN上可以使用的并发连接数量。

  取值范围：整数，>=-1，缺省值为-1，表示没有限制。

须知

为保证集群正常使用，connection limit的最小值是集群中CN的数目。在集群做ANALYZE时，其他CN节点会连接当前做ANALYZE的CN节点来同步元数据。例如集群中有3个CN节点，那么connection limit应该设置为>=3。

- **VALID BEGIN**

  设置角色生效的时间戳。如果省略了该子句，角色无有效开始时间限制。

- **VALID UNTIL**

  设置角色失效的时间戳。如果省略了该子句，角色无有效结束时间限制。

- **RESOURCE POOL**

  设置角色使用的resource pool名字，该名字属于系统表：pg_resource_pool。

- **USER GROUP 'groupuser'**

  创建一个user的子用户。

- **PERM SPACE**

  设置用户永久表存储空间限额。

  space_limit：永久表存储空间上限。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **TEMP SPACE**

  设置用户临时表存储空间限额。

  tmpspacelimit：临时表存储空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **SPILL SPACE**

  设置用户算子落盘空间限额。

  spillspacelimit：算子落盘空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **NODE GROUP**

  设置用户关联的逻辑集群名称。如果需要关联的逻辑集群名称包含大写字符或特殊字符，指定逻辑集群名称时需要加双引号。

- **IN ROLE**

  新角色立即拥有IN ROLE子句中列出的一个或多个现有角色拥有的权限。不推荐使用。

- **IN GROUP**

  IN GROUP是IN ROLE过时的拼法。不推荐使用。

- **ROLE**

  ROLE子句列出一个或多个现有的角色，它们将自动添加为这个新角色的成员，拥有新角色所有的权限。

- **ADMIN**

  ADMIN子句类似ROLE子句，不同的是ADMIN后的角色可以把新角色的权限赋给其他角色。

- **USER**

  USER子句是ROLE子句过时的拼法。

- **SYSID**

  SYSID子句将被忽略，无实际意义。

- **DEFAULT TABLESPACE**

  DEFAULT TABLESPACE子句将被忽略，无实际意义。

- **PROFILE**

  PROFILE子句将被忽略，无实际意义。

- **PGUSER**

该属性用于兼容开源Postgres的连接通讯，开源的Postgres客户端接口（推荐使用Postgres 9.2.19版本的相关客户端接口）可以使用具有该属性的数据库用户连接数据库。

须知

该属性只用于兼容连接过程，而由于本产品与Postgres的内核差异导致的不兼容，不在此属性控制范围内。

由于具有PGUSER属性的用户的认证方式与其他用户不同，开源客户端的报错信息可能导致数据库用户PGUSER属性被枚举，建议使用本产品自有的客户端。例如：

```
#normaluser是不具有PGUSER属性的用户，psql是Postgres的客户端工具
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser
psql: authentication method 10 not supported

#pguser用户是具有PGUSER属性的用户
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser
Password for user pguser:
```

- **AUTHINFO 'authinfo'**

  该属性用于指定角色认证类型，authinfo为类型说明字符串，大小写敏感。当前仅支持LDAP类型，对应的类型说明字符串为“ldap”。LDAP属于外部认证，故需要同时指定PASSWORD DISABLE。

  须知

  - authinfo可以加上LDAP认证的额外信息，比如LDAP认证中的fulluser，fulluser等同于ldapprefix+username+ldapsuffix。当authinfo为“ldap”，表示角色认证类型为LDAP，此时ldapprefix和ldapsuffix的信息由pg_hba.conf中匹配的记录提供。
  - ALTER ROLE时不允许用户切换认证类型，仅允许LDAP用户修改LDAP属性。

- **PASSWORD EXPIRATION period**

  声明该角色的登录密码过期天数，登录密码过期之前用户需要及时修改密码。登录密码过期后用户无法登录，需要请管理员设置新的登录密码后登录。

  取值范围：整数，-1~999。缺省值为-1，表示没有过期限制；0表示用户登录密码立即过期。

示例

创建一个角色，名为manager：

```
CREATE ROLE manager IDENTIFIED BY '{Password}';
```

创建一个角色，从2015年1月1日开始生效，到2026年1月1日失效：

```
CREATE ROLE miriam WITH LOGIN PASSWORD '{Password}' VALID BEGIN '2015-01-01' VALID UNTIL '2026-01-01';
```

创建一个角色，认证类型是LDAP，LDAP认证的其他信息由pg_hba.conf提供：

```
CREATE ROLE role1 WITH LOGIN AUTHINFO 'ldap' PASSWORD DISABLE;
```

创建一个角色，认证类型是LDAP，LDAP认证的fulluser信息在创建时指定，此时ldap大小写敏感，需要用单引号包含：

```
CREATE ROLE role2 WITH LOGIN AUTHINFO 'ldapcn=role2,cn=user,dc=lework,dc=com' PASSWORD DISABLE;
```

创建一个角色，登录密码有效期是30天：

```
CREATE ROLE role3 WITH LOGIN PASSWORD '{Password}' PASSWORD EXPIRATION 30;
```

相关链接

SET ROLE，**ALTER ROLE**，**DROP ROLE**，**GRANT**，**REVOKE**

12.46 CREATE SCHEMA

功能描述

创建模式。

访问命名对象时可以使用模式名作为前缀进行访问，如果无模式名前缀，则访问当前模式下的命名对象。创建命名对象时也可用模式名作为前缀修饰。

另外，CREATE SCHEMA可以包括在新模式中创建对象的子命令，这些子命令和那些在创建完模式后发出的命令没有任何区别。如果使用了AUTHORIZATION子句，则所有创建的对象都将被该用户所拥有。

注意事项

- 一个数据库可以包含一个或多个已命名的Schema，Schema又包含表及其他数据库对象，包括数据类型、函数、操作符等。同一对象名可以在不同的Schema中使用而不会引起冲突。例如，Schema1和Schema2都可以包含一个名为mytable的表。
- 和数据库不同，Schema不是严格分离的。用户根据其对Schema的权限，可以访问所连接数据库的Schema中的对象。进行Schema权限管理首先需要对数据库的权限控制进行了解。
- 不能创建以PG_为前缀的Schema名，该类Schema为数据库系统预留的。
- 在当前数据库中创建用户时，系统会在当前数据库中为新用户创建一个同名Schema。
- 只要用户对当前数据库有CREATE权限，就可以创建Schema。
- 系统管理员在普通用户同名Schema下创建的对象，所有者为Schema的同名用户（非系统管理员）。
- 通过未修饰的表名（名字中只含有表名，没有“Schema名”）引用表时，系统会通过search_path（搜索路径）来判断该表是哪个Schema下的表。pg_temp和pg_catalog始终会作为搜索路径顺序中的前两位，无论二者是否出现在search_path中，或者出现在search_path中的任何位置。search_path（搜索路径）是一个Schema名列表，在其中找到的第一个表就是目标表，如果没有找到则报错。某个表即使存在，如果它的Schema不在search_path中，依然会查找失败，在搜索路径中的第一个Schema叫做“当前Schema”。它是搜索时查询的第一个Schema，同时在没有声明Schema名时，新创建的数据库对象会默认存放在该Schema下。

语法格式

- 根据指定的名字创建模式：

```
CREATE SCHEMA schema_name
   [ AUTHORIZATION user_name ] [ WITH PERM SPACE 'space_limit'] [ schema_element [ ... ] ];
```

- 根据用户名创建模式：

```
CREATE SCHEMA AUTHORIZATION user_name [ WITH PERM SPACE 'space_limit'] [ schema_element [ ... ] ];
```

参数说明

- **schema_name**

  模式名字。

  须知

  - 模式名不能和当前数据库里其他的模式重名。
  - 模式的名字不可以“pg_”开头。

  取值范围：字符串，要符合标识符的命名规范。

- **AUTHORIZATION user_name**

  指定模式的所有者。当不指定schema_name时，把user_name当作模式名，此时user_name只能是角色名。

  取值范围：已存在的用户名/角色名。

- **WITH PERM SPACE 'space_limit'**

  指定模式的永久表存储空间上限。当不指定space_limit时，则不限制。

  取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。解析后的数值以K为单位，且范围不能够超过64比特表示的有符号整数，即1KB~9007199254740991KB。

- **schema_element**

  在模式里创建对象的SQL语句。目前仅支持CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE PARTITION、GRANT子句。

  子命令所创建的对象都被AUTHORIZATION子句指定的用户所拥有。

说明

如果当前搜索路径上的模式中存在同名对象时，需要明确指定引用对象所在的模式。可以通过命令SHOW SEARCH_PATH来查看当前搜索路径上的模式。

示例

创建一个角色role1：

```
CREATE ROLE role1 IDENTIFIED BY '{Password}';
```

为用户role1创建一个同名schema，子命令创建的表films和winners的拥有者为role1：

```
CREATE SCHEMA AUTHORIZATION role1
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
    SELECT title, release FROM films WHERE awards IS NOT NULL;
```

相关链接

ALTER SCHEMA，**DROP SCHEMA**

12.47 CREATE SEQUENCE

功能描述

向当前数据库里增加一个新的序列。序列的Owner为创建此序列的用户。

注意事项

- SEQUENCE是一个存放等差数列的特殊表，该表受DBMS控制。这个表没有实际意义，通常用于为行或者表生成唯一的标识符。
- 如果给出一个模式名，则该序列就在给定的模式中创建，否则会在当前模式中创建。序列名必须和同一个模式中的其他序列、表、索引、视图或外表的名字不同。
- 创建序列后，在表中使用序列的nextval()函数和generate_series(1,N)函数对表插入数据，请保证nextval的可调用次数大于等于N+1次，否则会因为generate_series()函数会调用N+1次而导致报错。
- 不支持在template1数据库中创建SEQUENCE。

语法格式

```
CREATE SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE | NOMINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE |
NOMAXVALUE]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE | NOCYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ];
```

参数说明

- **name**

  将要创建的序列名称。

  取值范围：仅可以使用小写字母（a~z）、 大写字母（A~Z），数字和特殊字符"#"，"_"，"$"的组合。

- **increment**

  指定序列的步长。一个正数将生成一个递增的序列，一个负数将生成一个递减的序列。

  缺省值：1。

- **MINVALUE minvalue | NO MINVALUE| NOMINVALUE**

  执行序列的最小值。如果没有声明minvalue或者声明了NO MINVALUE，则递增序列的缺省值为1，递减序列的缺省值为$-2^{63}-1$。

  NOMINVALUE等价于NO MINVALUE

- **MAXVALUE maxvalue | NO MAXVALUE| NOMAXVALUE**

  执行序列的最大值。如果没有声明maxvalue或者声明了NO MAXVALUE，则递增序列的缺省值为$2^{63}-1$，递减序列的缺省值为-1。

  NOMAXVALUE等价于NO MAXVALUE

- **start**

  指定序列的起始值。

  缺省值：对于递增序列为minvalue，递减序列为maxvalue。

- **cache**

  为了快速访问，而在内存中预先存储序列号的个数。一个缓存周期内，CN不再向GTM索取序列号，而是使用本地预先申请的序列号。

  缺省值为1，表示一次只能生成一个值，也就是没有缓存。

  说明

  - 不建议同时定义cache和maxvalue或minvalue。因为定义cache后不能保证序列的连续性，可能会产生空洞，造成序列号段浪费。
  - 建议cache值不要设置过大，否则会出现缓存序列号时（每个cache周期的第一个nextval）耗时过长的情况；同时建议cache值小于100000000。实际使用时应根据业务设置合理的cache值，既能保证快速访问，又不会浪费序列号。

- **CYCLE**

  用于使序列达到maxvalue或者minvalue后可循环并继续下去。

  如果声明了NO CYCLE，则在序列达到其最大值后任何对nextval的调用都会返回一个错误。

  NOCYCLE的作用等价于NO CYCLE。

  缺省值为NO CYCLE。

  若定义序列为CYCLE，则不能保证序列的唯一性。

- **OWNED BY**-

  将序列和一个表的指定字段进行关联。这样，在删除那个字段或其所在表的时候会自动删除已关联的序列。关联的表和序列的所有者必须是同一个用户，并且在同一个模式中。需要注意的是，通过指定OWNED BY，仅仅是建立了表的对应列和Sequence之间关联关系，并不会在插入数据时在该列上产生自增序列。

  缺省值为OWNED BY NONE，表示不存在这样的关联。

须知

通过OWNED BY创建的Sequence不建议用于其他表，如果希望多个表共享Sequence，该Sequence不应该从属于特定表。

示例

创建一个名为serial的递增序列，从101开始：

```
CREATE SEQUENCE serial
 START 101
 CACHE 20;
```

从序列中选出下一个数字：

```
SELECT nextval('serial');
 nextval
---------
     101
```

从序列中选出下一个数字：

```
SELECT nextval('serial');
 nextval
---------
     102
```

创建与表关联的序列：

```
CREATE TABLE customer_address
(
    ca_address_sk             integer               not null,
    ca_address_id             char(16)              not null,
    ca_street_number          char(10)                      ,
    ca_street_name            varchar(60)                   ,
    ca_street_type            char(15)                      ,
    ca_suite_number           char(10)                      ,
    ca_city                   varchar(60)                   ,
    ca_county                 varchar(30)                   ,
    ca_state                  char(2)                       ,
    ca_zip                    char(10)                      ,
    ca_country                varchar(20)                   ,
    ca_gmt_offset             decimal(5,2)                  ,
    ca_location_type          char(20)
) ;

CREATE SEQUENCE serial1
 START 101
 CACHE 20
OWNED BY customer_address.ca_address_sk;
```

使用serial创建主键自增序列表serial_table：

```
CREATE TABLE serial_table(a int, b serial);
INSERT INTO serial_table (a) VALUES (1),(2),(3);
SELECT * FROM serial_table ORDER BY b;
 a | b
---+---
 1 | 1
 2 | 2
 3 | 3
(3 rows)
```

相关链接

DROP SEQUENCE ALTER SEQUENCE

12.48 CREATE SERVER

功能描述

创建一个外部服务器。

外部服务器是存储HDFS集群信息、OBS服务器信息、DLI连接信息或其他同构集群信息的载体。

注意事项

默认只有系统管理员才可以创建外部服务器，否则需要对所使用的FOREIGN DATA WRAPPER授权，授权语法为：

```
GRANT USAGE ON FOREIGN DATA WRAPPER fdw_name TO username;
```

其中fdw_name为FOREIGN DATA WRAPPER的名字，username为创建SERVER的用户名。

语法格式

```
CREATE SERVER server_name
    FOREIGN DATA WRAPPER fdw_name
    OPTIONS ( { option_name ' value ' } [, ...] ) ;
```

参数说明

- **server_name**

  要创建的外部服务器的名称。服务器名称在数据库中必须唯一。

  取值范围：长度必须小于等于63。

- **FOREIGN DATA WRAPPER fdw_name**

  指定外部数据封装器的名字。

  取值范围：fdw_name是数据库初始化时系统创建的数据封装器，目前对于HDFS集群，fdw_name的名字可以是hdfs_fdw或者dfs_fdw，对于其他同构集群，fdw_name为gc_fdw。

- **OPTIONS ({ option_name ' value ' } [, ...])**

  用于指定外部服务器的各类参数，详细的参数说明如下所示：

  - address

    指定的OBS服务终端节点或HDFS集群的IP地址。

    OBS：OBS服务的终端节点（Endpoint）。

    HDFS：HDFS集群的元数据节点（NameNode）所在的IP地址以及端口，或者同构其他集群的CN的IP地址以及端口。

    为保证HA（High Availability），HDFS NameNode经常采用主备模式。主备NameNode的地址都需要加入到address值中。GaussDB(DWS)访问HDFS服务时，会动态查找当前处于active状态的主NameNode。

    若HDFS为联邦模式时，可将Router的地址都加入到address值中，GaussDB(DWS)访问HDFS服务时，会动态随机查找当前处于active状态的Router。

    说明

    - address option必须存在，若用于跨集群互联互通场景则只允许设置1个。
    - 当server类型为DLI时，address为DLI服务上数据所存储的OBS address。
    - 若HDFS为联邦模式时，即fed 'rbf'，address可设置为多组IP、port，对应为HDFS Router的address。

  - hdfscfgpath

    该参数仅支持type为HDFS时设置。

    用户通过配置hdfscfgpath参数来指定HDFS配置文件路径。GaussDB(DWS)会根据配置文件路径下的HDFS配置文件指定的连接配置方式，以及安全模式，来访问HDFS集群。非安全模式连接HDFS集群时，不支持数据传输加密。

    如果没有指定address选项，默认采用hdfscfgpath指定的配置文件中指定的address。

  - fed

    表示dfs_fdw连接的是HDFS为联邦模式。

    取值rbf，表示HDFS为联邦rbf方式。

    说明

    该参数8.1.2及以上版本支持。

  - encrypt

    是否对数据进行加密，该参数仅支持type为OBS时设置。默认值为off。

    取值范围：

  - on表示对数据进行加密。
  - off表示不对数据进行加密。
- access_key

  OBS访问协议对应的AK值（OBS云服务界面由用户获取），创建外表时AK值会保存到数据库的元数据表中。该参数仅支持type为OBS时设置。
- secret_access_key

  OBS访问协议对应的SK值（OBS云服务界面由用户获取），创建外表时SK值会加密保存到数据库的元数据表中。该参数仅支持type为OBS时设置。
- type

  表示dfs_fdw连接的类型。

  取值范围：
  - OBS表示连接的是OBS。
  - HDFS表示连接的是HDFS。
  - DLI表示连接的是DLI。
- dli_address

  DLI服务的终端节点，即endpoint。该参数仅支持type为DLI时设置。
- dli_access_key

  DLI访问协议对应的AK值（DLI云服务界面由用户获取），创建外表时AK值会保存到数据库的元数据表中。该参数仅支持type为DLI时设置。
- dli_secret_access_key

  DLI访问协议对应的SK值（DLI云服务界面由用户获取），创建外表时SK值会加密保存到数据库的元数据表中。该参数仅支持type为DLI时设置。
- dbname

  用于协同分析、跨集群互联互通，表示将要连接的远端集群的数据库名字。
- username

  用于协同分析、跨集群互联互通，表示将要连接的远端集群的用户名。
- password

  用于协同分析、跨集群互联互通，表示将要连接的远端集群的用户名密码。

  说明

  对于云下集群迁移到云上的场景，从云下集群导出的server配置中密码为密文，由于云上和云下集群加解密的密钥不同，如果直接在云上集群执行导出时的CREATE SERVER，会执行失败，报解密失败的错误。这种场景下需要将CREATE SERVER中的password手动修改成明文密码进行配置。
- syncsrv

  仅用于跨集群互联互通，表示数据同步过程中使用到的GDS服务，设置方式与GDS外表的location属性相同。

示例

建立一个hdfs_server，其中hdfs_fdw为数据库中存在的foreign data wrapper：

```
CREATE SERVER hdfs_server FOREIGN DATA WRAPPER HDFS_FDW OPTIONS
  (address '10.10.0.100:25000,10.10.0.101:25000',
```



```
  hdfscfgpath '/opt/hadoop_client/HDFS/hadoop/etc/hadoop',
  type 'HDFS'
) ;
```

建立一个obs_server，其中dfs_fdw为数据库中存在的foreign data wrapper：

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
CREATE SERVER obs_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (
  address 'obs.xxx.xxx.com',
  access_key 'xxxxxxxxx',
  secret_access_key 'yyyyyyyyyyyyy',
  type 'obs'
);
```

建立一个dli_server，其中dfs_fdw为数据库中存在的foreign data wrapper：

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
CREATE SERVER dli_server FOREIGN DATA WRAPPER DFS_FDW OPTIONS (
  address 'obs.xxx.xxx.com',
  access_key 'xxxxxxxxx',
  secret_access_key 'yyyyyyyyyyyyy',
  type 'dli',
  dli_address 'dli.xxx.xxx.com',
  dli_access_key 'xxxxxxxxx',
  dli_secret_access_key 'yyyyyyyyyyyyy'
);
```

建立另外一个同构集群的server，其中gc_fdw为数据库中存在的foreign data wrapper：

```
CREATE SERVER server_remote FOREIGN DATA WRAPPER GC_FDW OPTIONS
   (address '10.10.0.100:25000,10.10.0.101:25000',
  dbname 'test',
  username 'test',
  password '{Password}'
);
```

相关链接

ALTER SERVER DROP SERVER

12.49 CREATE SYNONYM

功能描述

创建一个同义词对象。同义词是数据库对象的别名，用于记录与其他数据库对象名间的映射关系，用户可以使用同义词访问关联的数据库对象。

注意事项

- 定义同义词的用户成为其所有者。
- 若指定模式名称，则同义词在指定模式中创建。否则在当前模式创建。
- 支持通过同义词访问的数据库对象包括：表、视图、函数和存储过程。
- 使用同义词时，用户需要具有对关联对象的相应权限。
- 支持使用同义词的DML语句包括：SELECT、INSERT、UPDATE、DELETE、EXPLAIN、CALL。
- 不支持关联函数或存储过程的CREATE SYNONYM语句出现在存储过程中，建议存储过程中使用系统表pg_synonym中已存在的同义词对象。

语法格式

```
CREATE [ OR REPLACE ] SYNONYM synonym_name
    FOR object_name;
```

参数说明

- **synonym**

  创建的同义词名字，可以带模式名。

  取值范围：字符串，要符合标识符的命名规范。

- **object_name**

  关联的对象名字，可以带模式名。

  取值范围：字符串，要符合标识符的命名规范。

  说明

  object_name可以是不存在的对象名称。

示例

创建模式ot和tpcds：

```
CREATE SCHEMA ot;
CREATE SCHEMA tpcds;
```

创建表ot.t1及其同义词t1：

```
CREATE TABLE ot.t1(id int, name varchar2(10)) DISTRIBUTE BY hash(id);
CREATE OR REPLACE SYNONYM t1 FOR ot.t1;
```

使用同义词t1：

```
SELECT * FROM t1;
INSERT INTO t1 VALUES (1, 'ada'), (2, 'bob');
UPDATE t1 SET t1.name = 'cici' WHERE t1.id = 2;
```

创建同义词v1及其关联视图ot.v_t1：

```
CREATE SYNONYM v1 FOR ot.v_t1;
CREATE VIEW ot.v_t1 AS SELECT * FROM ot.t1;
```

使用同义词v1：

```
SELECT * FROM v1;
```

创建重载函数ot.add及其同义词add：


```
CREATE OR REPLACE FUNCTION ot.add(a integer, b integer) RETURNS integer AS
$$
SELECT $1 + $2
$$
LANGUAGE sql;

CREATE OR REPLACE FUNCTION ot.add(a decimal(5,2), b decimal(5,2)) RETURNS decimal(5,2) AS
$$
SELECT $1 + $2
$$
LANGUAGE sql;

CREATE OR REPLACE SYNONYM add FOR ot.add;
```

使用同义词add：

```
SELECT add(1,2);
SELECT add(1.2,2.3);
```

创建存储过程ot.register及其同义词register：

```
CREATE PROCEDURE ot.register(n_id integer, n_name varchar2(10))
SECURITY INVOKER
AS
BEGIN
    INSERT INTO ot.t1 VALUES(n_id, n_name);
END;
/

CREATE OR REPLACE SYNONYM register FOR ot.register;
```

使用同义词register，调用存储过程：

```
CALL register(3,'mia');
```

相关链接

ALTER SYNONYM，**DROP SYNONYM**

12.50 CREATE TABLE

功能描述

在当前数据库中创建一个新的空白表。

该表由命令执行者所有，但系统管理员在普通用户同名schema下创建的表，表的所有者为schema的同名用户（非系统管理员）。

注意事项

- 列存表支持的数据类型请参考**列存表支持的数据类型**。
- 创建列存和HDFS分区表的数量建议不超过1000个。
- 表中的主键约束和唯一约束必须包含分布列。
- 如果在建表过程中数据库系统发生故障，系统恢复后可能无法自动清除之前已创建的、大小为0的磁盘文件。此种情况出现概率小，不影响数据库系统的正常运行。
- 列存表支持PARTIAL CLUSTER KEY、主键和唯一表级约束，不支持外键表级约束。

- 列存表的字段约束只支持NULL、NOT NULL和DEFAULT常量值。
- 列存表支持delta表，受表级参数enable_delta控制是否开启，受参数deltarow_threshold控制进入delta表的阈值。
- 冷热表仅支持列存分区表，依赖于可用的OBS服务。
- 冷热表仅支持默认表空间为default_obs_tbs，如需新增obs表空间可联系技术支持。

语法格式

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
  { ({ column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint [ ... ] ]
      | table_constraint
      | LIKE source_table [ like_option [...] ] }
      [, ... ])|
      LIKE source_table [ like_option [...] ] }
  [ WITH ( {storage_parameter = value} [, ... ] ) ]
  [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
  [ COMPRESS | NOCOMPRESS ]
  [ DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { HASH ( column_name [,...] ) } } ]
  [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
  [ COMMENT [=] 'text' ];
```

- 其中列约束column_constraint为：

  ```
  [ CONSTRAINT constraint_name ]
  { NOT NULL |
    NULL |
    CHECK ( expression ) |
    DEFAULT default_expr |
    COMMENT 'text' |
    UNIQUE index_parameters |
    PRIMARY KEY index_parameters }
  [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
  ```

- 其中列的压缩可选项compress_mode为：

  ```
  { DELTA | PREFIX | DICTIONARY | NUMSTR | NOCOMPRESS }
  ```

- 其中表约束table_constraint为：

  ```
  [ CONSTRAINT constraint_name ]
  { CHECK ( expression ) |
    UNIQUE ( column_name [, ... ] ) index_parameters |
    PRIMARY KEY ( column_name [, ... ] ) index_parameters |
    PARTIAL CLUSTER KEY ( column_name [, ... ] ) }
  [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
  ```

- 其中like选项like_option为：

  ```
  { INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS |
  PARTITION | RELOPTIONS | DISTRIBUTION | DROPCOLUMNS | ALL }
  ```

- 其中索引参数index_parameters为：

  ```
  [ WITH ( {storage_parameter = value} [, ... ] ) ]
  ```

参数说明

- **UNLOGGED**

  如果指定此关键字，则创建的表为非日志表。在非日志表中写入的数据不会被写入到预写日志中，这样就会比普通表快很多。但是非日志表在冲突、执行操作系统重启、强制重启、切断电源操作或异常关机后会被自动截断，会造成数据丢失的风险。非日志表中的内容也不会被复制到备服务器中。在非日志表中创建的索引也不会被自动记录。

  使用场景：非日志表不能保证数据的安全性，用户应该在确保数据已经做好备份的前提下使用，例如系统升级时进行数据的备份。

  故障处理：当异常关机等操作导致非日志表上的索引发生数据丢失时，用户应该对发生错误的索引进行重建。

须知

UNLOGGED表无主备机制，在系统故障或异常断点等情况下，会有数据丢失风险，不可用来存储基础数据。

- **GLOBAL | LOCAL**

  创建临时表时可以在TEMP或TEMPORARY前指定GLOBAL或LOCAL关键字。目前这两个关键字的设立，仅是为了兼容SQL标准，实际上无论指定GLOBAL还是LOCAL，GaussDB(DWS)都会创建本地临时表。

- **TEMPORARY | TEMP**

  如果指定TEMP或TEMPORARY关键字，则创建的表为临时表。临时表只在当前会话可见，本会话结束后会自动删除。因此，在除当前会话连接的CN以外的其他CN故障时，仍然可以在当前会话上创建和使用临时表。由于临时表只在当前会话创建，对于涉及对临时表操作的DDL语句，会产生DDL失败的报错。因此，建议DDL语句中不要对临时表进行操作。TEMP和TEMPORARY等价。

须知

- 临时表通过每个会话独立的以pg_temp开头的schema来保证只对当前会话可见，因此，不建议用户在日常操作中手动删除以pg_temp，pg_toast_temp开头的schema。
- 如果建表时不指定TEMPORARY/TEMP关键字，而指定表的schema为当前会话的pg_temp_开头的schema，则此表会被创建为临时表。

- **IF NOT EXISTS**

  指定IF NOT EXISTS时，若不存在同名表，则可以成功创建表。若已存在同名表，创建时不会报错，仅会提示该表已存在并跳过创建。

- **table_name**

  要创建的表名。

  表名长度不超过63个字符，以字母或下划线开头，可包含字母、数字、下划线、$、#。

  使用双引号括起来的表名可以包含空格和特殊字符，但不建议在表名中使用这些字符，因为这样可能会使表名难以引用和使用，而且不同的数据库兼容模式下可能对于这种表名的处理方式也有所不同。

- **column_name**

  新表中要创建的字段名。

  字段名长度不超过63个字符，以字母或下划线开头，可包含字母、数字、下划线、$、#。

- **data_type**

  字段的数据类型。

  说明

  在兼容Teradata或MySQL语法的数据库中，字段数据类型指定为DATE时同样返回为DATE类型，否则返回TIMESTAMP类型。

- **compress_mode**

表字段的压缩选项，当前仅对行存表有效。该选项指定表字段优先使用的压缩算法。

该压缩选项与列存表自适应压缩算法无关，后者为列存表内部数据存储采用的压缩算法，不支持用户指定压缩方式，详见**COMPRESSION**参数说明。

取值范围：DELTA、PREFIX、DICTIONARY、NUMSTR、NOCOMPRESS

- DELTA压缩仅支持长度为1-8字节的数据类型（ 0 < pg_type.typlen<=8 ）。
- PREFIX、NUMSTR压缩仅支持变长数据类型（ pg_type.typlen=-1 ）和NULL结尾的C字符串（ pg_type.typlen=-2 ）。

- **COLLATE collation**

  COLLATE子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。

- **LIKE source_table [like_option ...]**

  LIKE子句声明一个表，新表自动从这个表中继承所有字段名及其数据类型和非空约束。

  新表与源表之间在创建动作完毕之后是完全无关的。在源表做的任何修改都不会传播到新表中，并且也不可能在扫描源表的时候包含新表的数据。

  被复制的列和约束并不使用相同的名字进行融合。如果明确的指定了相同的名字或者在另外一个LIKE子句中，将会报错。

  - 源表上的字段缺省表达式只有在指定INCLUDING DEFAULTS时，才会复制到新表中。缺省是不包含缺省表达式的，即新表中的所有字段的缺省值都是NULL。
  - 源表上的CHECK约束仅在指定INCLUDING CONSTRAINTS时，会复制到新表中，而其他类型的约束永远不会复制到新表中。非空约束总是复制到新表中。此规则同时适用于表约束和列约束。
  - 如果指定了INCLUDING INDEXES，则源表上的索引也将在新表上创建，默认不建立索引。
  - 如果指定了INCLUDING STORAGE，则复制列的STORAGE设置会复制到新表中，默认情况下不包含STORAGE设置。
  - 如果指定了INCLUDING COMMENTS，则源表列、约束和索引的注释会复制到新表中。默认情况下，不复制源表的注释。
  - 如果指定了INCLUDING PARTITION，则源表的分区定义会复制到新表中，同时新表将不能再使用PARTITION BY子句。默认情况下，不复制源表的分区定义。
  - 如果指定了INCLUDING RELOPTIONS，则源表的存储参数（即源表的WITH子句）会复制到新表中。默认情况下，不复制源表的存储参数。
  - 如果指定了INCLUDING DISTRIBUTION，则源表的分布信息会复制到新表中，包括分布类型和分布列，同时新表将不能再使用DISTRIBUTE BY子句。默认情况下，不复制源表的分布信息。
  - 如果指定了INCLUDING DROPCOLUMNS，则源表被删除的列信息会被复制到新表中。默认情况下，不复制源表的删除列信息。
  - INCLUDING ALL包含了INCLUDING DEFAULTS、INCLUDING CONSTRAINTS、INCLUDING INDEXES、INCLUDING STORAGE、INCLUDING COMMENTS、INCLUDING PARTITION、INCLUDING RELOPTIONS、INCLUDING DISTRIBUTION和INCLUDING DROPCOLUMNS的内容。
  - 如果指定了EXCLUDING，则表示不包括指定的参数。

- 如果是OBS冷热表，INCLUDING PARTITION后新表所有分区均为本地热分区。

须知

- 如果源表包含serial、bigserial、smallseriral类型，或者源表字段的默认值是Sequence，且Sequence属于源表（通过CREATE SEQUENCE ... OWNED BY创建），这些Sequence不会关联到新表中，新表中会重新创建属于自己的Sequence。这和之前版本的处理逻辑不同。如果用户希望源表和新表共享Sequence，需要首先创建一个共享的Sequence（避免使用OWNED BY），并配置为源表字段默认值，这样创建的新表会和源表共享该Sequence。
- 不建议将其他表私有的Sequence配置为源表字段的默认值，尤其是其他表只分布在特定的NodeGroup上，这可能导致CREATE TABLE ... LIKE执行失败。另外，如果源表配置其他表私有的Sequence，当该表删除时Sequence也会连带删除，这样源表的Sequence将不可用。如果用户希望多个表共享Sequence，建议创建共享的Sequence。

- **WITH ({ storage_parameter = value } [, ...])**

  这个子句为表或索引指定一个可选的存储参数。

  说明

  使用任意精度类型Numeric定义列时，建议指定精度p以及刻度s。在不指定精度和刻度时，会按输入的显示出来。

  参数的详细描述如下所示。

  - FILLFACTOR

    一个表的填充因子（fillfactor）是一个介于10和100之间的百分数。100（完全填充）是默认值。如果指定了较小的填充因子，INSERT操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得UPDATE有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为100是合适的选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数对于列存表没有意义。

    取值范围：10~100

  - ORIENTATION

    指定表数据的存储方式，即行存方式、列存方式，该参数设置成功后就不再支持修改。

    取值范围：

    - ROW，表示表的数据将以行式存储。

      行存储适合于OLTP业务，此类型的表上交互事务比较多，一次交互会涉及表中的多个列，用行存查询效率较高。

    - COLUMN，表示表的数据将以列式存储。

      列存储适合于数据仓库业务，此类型的表上会做大量的汇聚计算，且涉及的列操作较少。

    默认值：ROW，即行存方式。

说明

8.1.3及以上集群版本中新增GUC参数default_orientation（默认值为row），该参数设置创建表时若不指定存储方式，可根据其参数的取值（row, column, column enabledelta）创建对应的行存表，列存表或开启delta表的列存表。

- COMPRESSION

  指定表数据的压缩级别，它决定了表数据的压缩比以及压缩时间。一般来讲，压缩级别越高，压缩比也越大，压缩时间也越长；反之亦然。实际压缩比取决于加载的表数据的分布特征。

  取值范围：

  列存表的有效值为YES/NO和LOW/MIDDLE/HIGH，默认值为LOW。当设置为YES时，压缩级别默认为LOW。

  说明

  - 暂不支持行存表压缩功能。

  GaussDB(DWS)内部提供如下压缩算法。

  表 12-26 列存压缩算法

COMPRESSION	NUMERIC	STRING	INT
LOW	delta压缩+RLE压缩	lz4压缩	delta压缩（RLE可选）
MIDDLE	delta压缩+RLE压缩+lz4压缩	dict压缩或lz4压缩	delta压缩或lz4压缩（RLE可选）
HIGH	delta压缩+RLE压缩+zlib压缩	dict压缩或zlib压缩	delta压缩或zlib压缩（RLE可选）

- COMPRESSLEVEL

  指定表数据同一压缩级别下的不同压缩水平，它决定了同一压缩级别下表数据的压缩比以及压缩时间。对同一压缩级别进行了更加详细的划分，为用户选择压缩比和压缩时间提供了更多的空间。总体来讲，此值越大，表示同一压缩级别下压缩比越大，压缩时间越长；反之亦然。该参数只对列存表有效。

  取值范围：0~3，默认值为0。

- MAX_BATCHROW

  指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

  取值范围：10000~60000

  默认值：60,000

- PARTIAL_CLUSTER_ROWS

  指定了在数据加载过程中进行将局部聚簇存储的记录数目。该参数只对列存表有效。

  取值范围：600000~2147483647

  默认值：4,200,000

- enable_delta

  指定了在列存表是否开启delta表。该参数只对列存表有效。

  默认值：off

- DELTAROW_THRESHOLD

  指定列存表导入时小于多少行的数据进入delta表，只在表级参数enable_delta开启时生效。该参数只对列存表有效。

  取值范围：0～60000，默认值为6000

- COLVERSION

  指定列存存储格式的版本，支持不同存储格式版本之间的切换。

  取值范围：

  1.0：列存表的每列以一个单独的文件进行存储，文件名以relfilenode.C1.0、relfilenode.C2.0、relfilenode.C3.0等命名。

  2.0：列存表的每列合并存储在一个文件中，文件名以relfilenode.C1.0命名。

  默认值：2.0

  需注意，OBS冷热表仅支持COLVERSION为2.0格式。

  说明

  - 8.1.0集群版本该参数默认值为1.0，8.1.1及以上集群版本该参数默认值为2.0，若集群版本由8.1.0升级至8.1.1或以上版本，该参数默认值也会由1.0变为2.0。
  - 在建列存表时，选择COLVERSION=2.0，相比于1.0存储格式，在以下场景中性能有明显提升：
    1. 创建列存宽表场景下，建表时间显著减少。
    2. roach备份数据场景下，备份时间显著减少。
    3. build、catch up耗时显著减少。
    4. 占用磁盘空间大小显著减少。

- SKIP_FPI_HINT

  顺序扫描过程中，若需要写FPW(full page writes)日志时，该参数控制是否跳过设置HintBits操作。

  默认值：false

  说明

  设置SKIP_FPI_HINT=true时，在对某表执行checkpoint操作后，若对该表进行顺序扫描，将不再产生Xlog。适用于查询次数较少的中间表，有效减少Xlog的大小，提升查询性能。

- **ON COMMIT { PRESERVE ROWS | DELETE ROWS }**

  ON COMMIT选项决定在事务中执行创建临时表操作，当事务提交时，此临时表的后续操作。

  - PRESERVE ROWS（缺省值）：提交时不对临时表做任何操作，临时表及其表数据保持不变。
  - DELETE ROWS：提交时删除临时表中数据。

- **COMPRESS | NOCOMPRESS**

  创建新表时，需要在CREATE TABLE语句中指定关键字COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字NOCOMPRESS则不对表进行压缩。

缺省值：NOCOMPRESS，即不对元组数据进行压缩。

- **DISTRIBUTE BY**

  指定表如何在节点之间分布或者复制。

  取值范围：

  - REPLICATION：表的每一行存在所有数据节点（DN）中，即每个数据节点都有完整的表数据。
  - ROUNDROBIN：表的每一行被依次发送给各个DN，在这种分布策略下可以保证数据分布不会存在倾斜，但是因为数据分布节点是随机的，导致这类表在计算时会更大概率的触发此表的重分布。各列倾斜都比较严重的大表推荐使用此种分布策略。（ROUNDROBIN仅8.1.2及以上版本支持）
  - HASH（column_name）：对指定的列进行Hash，通过映射，把数据分布到指定DN。

    说明

    - 当指定DISTRIBUTE BY HASH (column_name)参数时，创建主键和唯一索引必须包含“ column_name”列。
    - 当被参照表指定DISTRIBUTE BY HASH (column_name)参数时，参照表的外键必须包含“ column_name”列。
    - 如果TO GROUP指定为复制表节点组（8.1.2及以上版本支持），DISTRIBUTE BY必须指定为REPLICATION。如果没有指定DISTRIBUTE BY，创建的表会自动设置为复制表。
    - 实时数仓（单机部署）由于只有单DN，因此分布规则会被忽略，也不支持针对分布规则的修改。

  默认值：由GUC参数default_distribution_mode控制。

  - 当default_distribution_mode=roundrobin时，DISTRIBUTE BY的默认值按如下规则选取：
    i. 若建表时包含主键/唯一约束，则选取HASH分布，分布列为主键/唯一约束对应的列。
    ii. 若建表时不包含主键/唯一约束，则选取ROUNDROBIN分布。
  - 当default_distribution_mode=hash时，DISTRIBUTE BY的默认值按如下规则选取：
    i. 若建表时包含主键/唯一约束，则选取HASH分布，分布列为主键/唯一约束对应的列。
    ii. 若建表时不包含主键/唯一约束，但存在数据类型支持作分布列的列，则选取HASH分布，分布列为第一个数据类型支持作分布列的列。
    iii. 若建表时不包含主键/唯一约束，也不存在数据类型支持作分布列的列，选取ROUNDROBIN分布。

  以下数据类型支持作为分布列：

  - INTEGER TYPES：TINYINT，SMALLINT，INT，BIGINT，NUMERIC/DECIMAL
  - CHARACTER TYPES：CHAR，BPCHAR，VARCHAR，VARCHAR2，NVARCHAR2，TEXT
  - DATE/TIME TYPES：DATE，TIME，TIMETZ，TIMESTAMP，TIMESTAMPTZ，INTERVAL，SMALLDATETIME

说明

在建表时，选择分布列和分区键可对SQL查询性能产生重大影响。因此，需要根据一定策略选择合适的分布列和分区键。

- 选择合适的分布列

  对于采用散列（Hash）方式的数据分布表，一个合适的分布列应将一个表内的数据，均匀分散存储在多个DN内，避免出现数据倾斜现象（即多个DN内数据分布不均）。请按照如下原则判定合适的分布列：

  1. 判断是否已发生数据倾斜现象。

     连接数据库，执行如下语句，查看各DN内元组数目。命令中的斜体部分tablename，请填入待分析的表名。

     SELECT a.count,b.node_name FROM (SELECT count(*) AS count,xc_node_id FROM *tablename* **GROUP BY xc_node_id) a, pgxc_node b WHERE a.xc_node_id=b.node_id ORDER BY a.count DESC;**

     如果各DN内元组数目相差较大（如相差数倍、数十倍），则表明已发生数据倾斜现象，请按照下面原则调整分布列。

  2. 重新选择分布列，可通过ALTER TABLE语句调整分布列，选择原则如下：

     分布列的列值应比较离散，以便数据能够均匀分布到各个DN。例如，考虑选择表的主键为分布列，如在人员信息表中选择身份证号码为分布列。

     在满足上面原则的情况下，考虑选择查询中的连接条件为分布列，以便Join任务能够下推到DN中执行，且减少DN之间的通信数据量。

  3. 如果找不到一个合适的分布列，使数据能够均匀分布到各个DN，那么可以考虑使用REPLICATION或ROUNDROBIN的数据分布方式。由于REPLICATION的数据分布方式会在每个DN中存放完整的数据，因此在表较大且找不到合适的分布列时，推荐使用ROUNDROBIN的数据分布方式。（ROUNDROBIN分布方式8.1.2及以上版本支持）

- 选择合适的分区键

  数据分区功能，可根据表的一列或者多列，将要插入表的记录分为若干个范围（这些范围在不同的分区里没有重叠）。然后为每个范围创建一个分区，用来存储相应的数据。

  调整分区键，使每次查询结果尽可能存储在相同或者最少的分区内（称为“分区剪枝”），通过获取连续I/O大幅度提升查询性能。

  实际业务中，经常将时间作为查询对象的过滤条件，因此，可考虑选择时间列为分区键，键值范围可根据总数据量、一次查询数据量调整。

- **TO { GROUP groupname | NODE (nodename [, ...]) }**

  TO GROUP指定创建表所在的Node Group，目前不支持hdfs表使用。TO NODE主要供内部扩容工具使用，一般用户不应该使用。

  在逻辑集群模式下，如果不指定TO GROUP，表默认会创建在逻辑集群用户关联的节点组中；如果用户没有管理逻辑集群（例如管理员用户或其他普通用户），表默认会创建在第一个逻辑集群中（pgxc_group中oid最小的逻辑集群是第一个逻辑集群）。

  如果TO GROUP指定的节点组是复制表节点组，表将创建在所有CN和DN节点上，但复制表数据将只分布在复制表节点组包含的DN节点上。

- **COMMENT [=] 'text'**

  COMMENT子句可在创建表时指定表注释。

- **CONSTRAINT constraint_name**

  列约束或表约束的名字。可选的约束子句用于声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。

  定义约束有两种方法：

  – 列约束：作为一个列定义的一部分，仅影响该列。

- 表约束：不和某个列绑在一起，可以作用于多个列。

- **NOT NULL**

  字段值不允许为NULL。

- **NULL**

  字段值允许为NULL ，这是缺省值。

  这个子句只是为和非标准SQL数据库兼容。不建议使用。

- **CHECK (expression)**

  CHECK约束声明一个布尔表达式，每次要插入的新行或者要更新的行的新值必须使表达式结果为真或未知才能成功，否则会抛出一个异常并且不会修改数据库。

  声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。

  说明

  expression表达式中，如果存在“<>NULL”或“！=NULL”，这种写法是无效的，需要写成“is NOT NULL”。

- **DEFAULT default_expr**

  DEFAULT子句给字段指定缺省值。该数值可以是任何不含变量的表达式(不允许使用子查询和对本表中的其他字段的交叉引用)。缺省表达式的数据类型必须和字段类型匹配。

  缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为NULL 。

- **COMMENT 'text'**

  COMMENT子句可以指定列的注释。

- **UNIQUE index_parameters**

  UNIQUE (column_name [, ...]) index_parameters

  UNIQUE约束表示表里的一个字段或多个字段的组合必须在全表范围内唯一。

  对于唯一约束，NULL被认为是互不相等的。

  说明

  如果没有声明DISTRIBUTE BY REPLICATION，则唯一约束的列集合中必须包含分布列。

- **PRIMARY KEY index_parameters**

  PRIMARY KEY (column_name [, ...]) index_parameters

  主键约束声明表中的一个或者多个字段只能包含唯一的非NULL值。

  一个表只能声明一个主键。

  说明

  如果没有声明DISTRIBUTE BY REPLICATION，则主键约束的列集合中必须包含分布列。

- **DEFERRABLE | NOT DEFERRABLE**

  这两个关键字设置该约束是否可推迟。一个不可推迟的约束将在每条命令之后马上检查。可推迟约束可以推迟到事务结尾使用SET CONSTRAINTS命令检查。缺省是NOT DEFERRABLE。目前，只有行存的UNIQUE约束和主键约束可以接受这个子句。所有其他约束类型都是不可推迟的。

- **PARTIAL CLUSTER KEY**

  局部聚簇存储，列存表导入数据时按照指定的列(单列或多列)，进行局部排序。

- **INITIALLY IMMEDIATE | INITIALLY DEFERRED**

  如果约束是可推迟的，则这个子句声明检查约束的缺省时间。

  - 如果约束是INITIALLY IMMEDIATE（缺省），则在每条语句执行之后就立即检查它；
  - 如果约束是INITIALLY DEFERRED ，则只有在事务结尾才检查它。

  约束检查的时间可以用SET CONSTRAINTS命令修改。

示例

为表定义唯一列约束：

```
DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE CUSTOMER
(
   C_CUSTKEY     BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY  ,
   C_NAME        VARCHAR(25)  ,
   C_ADDRESS     VARCHAR(40)  ,
   C_NATIONKEY   INT          ,
   C_PHONE       CHAR(15)     ,
   C_ACCTBAL     DECIMAL(15,2)
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

为表定义主键表约束，可以在表的一列或多列上定义主键表约束：

```
DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE CUSTOMER
(
   C_CUSTKEY     BIGINT       ,
   C_NAME        VARCHAR(25)  ,
   C_ADDRESS     VARCHAR(40)  ,
   C_NATIONKEY   INT          ,
   C_PHONE       CHAR(15)     ,
   C_ACCTBAL     DECIMAL(15,2)   ,
   CONSTRAINT C_CUSTKEY_KEY PRIMARY KEY(C_CUSTKEY,C_NAME)
)
DISTRIBUTE BY HASH(C_CUSTKEY,C_NAME);
```

定义CHECK列约束：

```
DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE CUSTOMER
(
   C_CUSTKEY     BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY  ,
   C_NAME        VARCHAR(25)  ,
   C_ADDRESS     VARCHAR(40)  ,
   C_NATIONKEY   INT NOT NULL  CHECK (C_NATIONKEY > 0)
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

定义CHECK表约束：

```
DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE CUSTOMER
(
   C_CUSTKEY     BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY  ,
   C_NAME        VARCHAR(25)      ,
   C_ADDRESS     VARCHAR(40)      ,
   C_NATIONKEY   INT              ,
   CONSTRAINT C_CUSTKEY_KEY2 CHECK(C_CUSTKEY > 0 AND C_NAME <> '')
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

创建列存表指定存储格式和压缩方式：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH,COLVERSION=2.0)
DISTRIBUTE BY HASH (ca_address_sk);
```

使用DEFAULT为列W_STATE声明默认值：

```
DROP TABLE IF EXISTS warehouse_t;
CREATE TABLE warehouse_t
(
    W_WAREHOUSE_SK          INTEGER               NOT NULL,
    W_WAREHOUSE_ID          CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)    UNIQUE DEFERRABLE,
    W_WAREHOUSE_SQ_FT       INTEGER                        ,
    W_COUNTY                VARCHAR(30)                    ,
    W_STATE                 CHAR(2)           DEFAULT 'GA',
    W_ZIP                   CHAR(10)
);
```

以like方式创建一个表CUSTOMER_bk：

```
DROP TABLE IF EXISTS CUSTOMER_bk;
CREATE TABLE CUSTOMER_bk (LIKE CUSTOMER INCLUDING ALL);
```

相关链接

ALTER TABLE，**RENAME TABLE**，**DROP TABLE**

12.51 CREATE TABLE AS

功能描述

根据查询结果创建表。

CREATE TABLE AS创建一个表并且用来自SELECT命令的结果填充该表，该表的字段和SELECT输出字段的名字及数据类型相关。不过用户可以通过明确地给出一个字段名字列表来覆盖SELECT输出字段的名字。

CREATE TABLE AS对源表进行一次查询，然后将数据写入新表中，而查询视图结果会根据源表的变化而有所改变。相比之下，每次做查询的时候，视图都重新计算定义它的SELECT语句。

注意事项

- 分区表不能采用此方式进行创建。
- 如果在建表过程中数据库系统发生故障，系统恢复后可能无法自动清除之前已创建的、大小非0的磁盘文件。此种情况出现概率小，不影响数据库系统的正常运行。

语法格式

```
CREATE [ UNLOGGED ] TABLE table_name
    [ (column_name [, ...] ) ]
```

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ COMPRESS | NOCOMPRESS ]

[ DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { [HASH ] ( column_name ) } } ]

[ COMMENT [=] 'text' ]
AS query
[ WITH [ NO ] DATA ];
```

参数说明

- **UNLOGGED**

  指定表为非日志表。在非日志表中写入的数据不会被写入到预写日志中，这样就会比普通表快很多。但是，它也是不安全的，非日志表在冲突或异常关机后会被自动删截。非日志表中的内容也不会被复制到备用服务器中。在该类表中创建的索引也不会被自动记录。

  - 使用场景：非日志表不能保证数据的安全性，用户应该在确保数据已经做好备份的前提下使用，例如系统升级时进行数据的备份。
  - 故障处理：当异常关机等操作导致非日志表上的索引发生数据丢失时，用户应该对发生错误的索引进行重建。

  注意

  UNLOGGED表无主备机制，在系统故障或异常断点等情况下，会有数据丢失风险，不可用来存储基础数据。

- **table_name**

  要创建的表名。

  取值范围：字符串，要符合标识符的命名规范。

- **column_name**

  新表中要创建的字段名。

  取值范围：字符串，要符合标识符的命名规范。

- **WITH (storage_parameter [= value] [, ...])**

  这个子句为表或索引指定一个可选的存储参数。参数的详细说明如下所示。

  - FILLFACTOR

    一个表的填充因子（fillfactor）是一个介于10和100之间的百分数。如果指定了较小的填充因子，INSERT操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得UPDATE有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表，将填充因子设为100是合适的选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数只对行存表有效。

    取值范围：10~100

    默认值：100，即完全填充。

  - ORIENTATION

    取值范围：

    COLUMN：表的数据将以列式存储。

    ROW（缺省值）：表的数据将以行式存储。

  - COMPRESSION

指定表数据的压缩级别，它决定了表数据的压缩比以及压缩时间。一般来讲，压缩级别越高，压缩比也越大，压缩时间也越长；反之亦然。实际压缩比取决于加载的表数据的分布特征。

取值范围：

列存表的有效值为YES/NO和LOW/MIDDLE/HIGH，默认值为LOW。

说明

暂不支持行存表压缩功能。

- MAX_BATCHROW

  指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

  取值范围：10000~60000

  默认值：60000

- PARTIAL_CLUSTER_ROWS

  指定了在数据加载过程中进行将局部聚簇存储的记录数目。该参数只对列存表有效。

  取值范围：600000~2147483647

  默认值：4,200,000

- enable_delta

  指定了在列存表是否开启delta表。该参数只对列存表有效。

  默认值：off

- COLVERSION

  指定列存存储格式的版本，支持不同存储格式版本之间的切换。

  取值范围：

  1.0：列存表的每列以一个单独的文件进行存储，文件名以relfilenode.C1.0、relfilenode.C2.0、relfilenode.C3.0等命名。

  2.0：列存表的每列合并存储在一个文件中，文件名以relfilenode.C1.0命名

  默认值：2.0

  说明

  在建列存表时选择COLVERSION=2.0，相比于1.0存储格式，在以下场景中性能有明显提升：

  1. 创建列存宽表场景下，建表时间显著减少。
  2. roach备份数据场景下，备份时间显著减少。
  3. build、catch up耗时显著减少。
  4. 占用磁盘空间大小显著减少。

- SKIP_FPI_HINT

  顺序扫描过程中，若需要写FPW(full page writes)日志时，该参数控制是否跳过设置HintBits操作。

  默认值：false

  说明

  设置SKIP_FPI_HINT=true时，在对某表执行checkpoint操作后，若对该表进行顺序扫描，将不再产生Xlog。适用于查询次数较少的中间表，有效减少Xlog的大小，提升查询性能。

- **COMPRESS / NOCOMPRESS**

  创建一个新表时，需要在创建表语句中指定关键字COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字NOCOMPRESS则不对表进行压缩。

  缺省值：NOCOMPRESS，即不对元组数据进行压缩。

- **DISTRIBUTE BY**

  指定表如何在节点之间分布或者复制。

  - REPLICATION：表的每一行存在所有数据节点(DN)中，即每个数据节点都有完整的表数据。
  - ROUNDROBIN：表的每一行被依次发送给各个DN，在这种分布策略下可以保证数据分布不会存在倾斜，但是因为数据分布节点是随机的，导致这类表在计算时会更大概率的触发此表的重分布。各列倾斜都比较严重的大表推荐使用此种分布策略。（ ROUNDROBIN仅8.1.2及以上版本支持 ）
  - HASH（column_name ）：对指定的列进行Hash，通过映射，把数据分布到指定DN。

  须知

  - 当指定DISTRIBUTE BY HASH (column_name)参数时，创建主键和唯一索引必须包含“ column_name”列。
  - 当被参照表指定DISTRIBUTE BY HASH (column_name)参数时，参照表的外键必须包含“ column_name”列。

  默认值：由GUC参数default_distribution_mode控制。

  - 当default_distribution_mode=roundrobin时，DISTRIBUTE BY的默认值按如下规则选取：
    i. 若建表时包含主键/唯一约束，则选取HASH分布，分布列为主键/唯一约束对应的列。
    ii. 若建表时不包含主键/唯一约束，则选取ROUNDROBIN分布。
  - 当default_distribution_mode=hash时，DISTRIBUTE BY的默认值按如下规则选取：
    i. 若建表时包含主键/唯一约束，则选取HASH分布，分布列为主键/唯一约束对应的列。
    ii. 若建表时不包含主键/唯一约束，但存在数据类型支持作分布列的列，则选取HASH分布，分布列为第一个数据类型支持作分布列的列。
    iii. 若建表时不包含主键/唯一约束，也不存在数据类型支持作分布列的列，选取ROUNDROBIN分布。

  以下数据类型支持作为分布列：

  - INTEGER TYPES：TINYINT，SMALLINT，INT，BIGINT，NUMERIC/DECIMAL
  - CHARACTER TYPES：CHAR，BPCHAR，VARCHAR，VARCHAR2，NVARCHAR2，TEXT
  - DATE/TIME TYPES：DATE，TIME，TIMETZ，TIMESTAMP，TIMESTAMPTZ，INTERVAL，SMALLDATETIME

- **COMMENT [=] 'text'**

  COMMENT子句可以在创建表时指定表注释。

- **AS query**

  一个SELECT VALUES命令或者一个运行预备好的SELECT或VALUES查询的EXECUTE命令。

- **[WITH [NO] DATA]**

  创建表时，是否也插入查询到的数据。默认是要数据，选择“NO”参数时，则不要数据。

示例

创建表CUSTOMER：

```
DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE CUSTOMER
(
   C_CUSTKEY     BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY ,
   C_NAME        VARCHAR(25) ,
   C_ADDRESS     VARCHAR(40) ,
   C_NATIONKEY   INT NOT NULL  CHECK (C_NATIONKEY > 0)
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

创建一个表store_returns_t1并插入CUSTOMER表中C_CUSTKEY字段中大于4795的数值：

```
CREATE TABLE store_returns_t1 AS SELECT * FROM CUSTOMER WHERE C_CUSTKEY > 4795;
```

使用store_returns复制一个新表store_returns_t2：

```
CREATE TABLE store_returns_t2 AS table store_returns;
```

相关链接

CREATE TABLE，**SELECT**

12.52 CREATE TABLE PARTITION

功能描述

创建分区表。逻辑上的一张表根据某种方案分成几张物理块进行存储，这张逻辑上的表称之为分区表，物理块称之为分区。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的。

常见的分区策略包括：范围分区（Range Partitioning）、哈希分区（Hash Partitioning）、列表分区（List Partitioning）和数值分区（Value Partitioning）。

范围分区是根据表的一列或者多列，将要插入表的记录分为若干个范围，这些范围在不同的分区里没有重叠。为每个范围创建一个分区，用来存储相应的数据。

范围分区策略是指记录插入分区的方式，根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则给出报错和提示信息。这是最常用的分区策略。目前范围分区仅支持范围分区策略。

列表分区是根据表的一列，将要插入表的记录通过每一个分区中出现的键值划分到对应的分区中，这些键值在不同的分区里没有重叠。为每组键值创建一个分区，用来存储相应的数据。

列表分区策略是根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则给出报错和提示信息。

常见的分区策略都是按照某一列或者某几列定义一些数据分布范围，然后每个分区承载一个范围的数据，这些列称之为分区键。

说明

- 目前GaussDB(DWS)行存表、列存表仅支持范围分区和列表分区。
- 列表分区（List Partitioning）仅8.1.3及以上集群版本支持。

分区的优势

- 某些类型的查询性能可以得到极大提升，特别是表中访问率较高的行位于一个单独分区或少数几个分区上的情况下。分区可以减少数据的搜索空间，提高数据访问效率。
- 当查询或更新一个分区的大部分记录时，连续扫描对应分区而不是访问整个表可以获得巨大的性能提升。
- 如果需要大量加载或者删除的记录位于单独的分区上，则可以通过直接读取或删除那个分区以获得巨大的性能提升，同时还可以避免由于大量DELETE导致的VACUUM超载（仅范围分区）。

注意事项

有限地支持唯一约束和主键约束，即唯一约束和主键约束的约束键必须包含所有分区键。

语法格式

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
( [
   { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
   | table_constraint
   | LIKE source_table [ like_option [...] ] }[, ... ]
] )
   [ WITH ( {storage_parameter = value} [, ... ] ) ]
   [ COMPRESS | NOCOMPRESS ]
   [ DISTRIBUTE BY { REPLICATION | ROUNDROBIN | { [ HASH ] ( column_name ) } } ]
   [ TO { GROUP groupname | NODE ( nodename [, ... ] ) } ]
   PARTITION BY {
      {VALUES (partition_key)} |
      {RANGE (partition_key) ( partition_less_than_item [, ... ] )} |
      {RANGE (partition_key) ( partition_start_end_item [, ... ] )} |
      {LIST (partition_key) (list_partition_item [, ...])}
   } [ { ENABLE | DISABLE } ROW MOVEMENT ];
```

- 列约束column_constraint：

  ```
  [ CONSTRAINT constraint_name ]
  { NOT NULL |
    NULL |
    CHECK ( expression ) |
    DEFAULT default_expr |
    UNIQUE index_parameters |
    PRIMARY KEY index_parameters }
  [ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
  ```


- 表约束table_constraint：
```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters}
[ DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```
- like选项like_option：
```
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS |
RELOPTIONS | DISTRIBUTION | ALL }
```
- 索引存储参数index_parameters：
```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
```
- partition_less_than_item：
```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE } )
```
- partition_start_end_item：
```
PARTITION partition_name {
      {START(partition_value) END (partition_value) EVERY (interval_value)} |
      {START(partition_value) END ({partition_value | MAXVALUE})} |
      {START(partition_value)} |
      {END({partition_value | MAXVALUE})}
}
```
- list_partition_item：
```
PARTITION partition_name VALUES ( { (partition_value) [, ...] | DEFAULT } )
```

参数说明

- **IF NOT EXISTS**

  如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。
- **partition_table_name**

  分区表的名称。

  取值范围：字符串，要符合标识符的命名规范。
- **column_name**

  新表中要创建的字段名。

  取值范围：字符串，要符合标识符的命名规范。
- **data_type**

  字段的数据类型。
- **COLLATE collation**

  COLLATE子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。

  可排列的数据类型有char、varchar、text、nchar、nvarchar。
- **CONSTRAINT constraint_name**

  列约束或表约束的名字。可选的约束子句用于声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。

  定义约束有两种方法：

  - 列约束：作为一个列定义的一部分，仅影响该列。
  - 表约束：不和某个列绑在一起，可以作用于多个列。
- **LIKE source_table [like_option ...]**

  LIKE子句声明一个表，新表自动从声明的表中继承所有字段名及其数据类型和非空约束。

新表与原来的表之间在创建动作完毕之后是完全无关的。在源表做的任何修改都不会传播到新表中，并且也不可能在扫描源表的时候包含新表的数据。

字段缺省表达式只有在声明了INCLUDING DEFAULTS之后才会包含进来。缺省是不包含缺省表达式的，即新表中所有字段的缺省值都是NULL。

非空约束将总是复制到新表中，CHECK约束则仅在指定了INCLUDING CONSTRAINTS的时候才复制，而其他类型的约束则永远也不会被复制。此规则同时适用于表约束和列约束。

被复制的列和约束并不使用相同的名字进行融合。如果明确的指定了相同的名字或者在另外一个LIKE子句中，将会报错。

- 如果指定了INCLUDING INDEXES，则源表上的索引也将在新表上创建，默认不建立索引。
- 如果指定了INCLUDING STORAGE，则复制列的STORAGE设置也将被复制，默认情况下不包含STORAGE设置。
- 如果指定了INCLUDING COMMENTS，则源表列、约束和索引的注释也会被复制过来。默认情况下，不复制源表的注释。
- 如果指定了INCLUDING RELOPTIONS，则源表的存储参数（即源表的WITH子句）也将复制至新表。默认情况下，不复制源表的存储参数。
- 如果指定了INCLUDING DISTRIBUTION，则新表将复制源表的分布信息，包括分布类型和分布列，同时新表将不能再使用DISTRIBUTE BY子句。默认情况下，不复制源表的分布信息。
- INCLUDING ALL是INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STORAGE INCLUDING COMMENTS INCLUDING RELOPTIONS INCLUDING DISTRIBUTION的简写形式。

- **WITH (storage_parameter [= value] [, ...])**

  这个子句为表或索引指定一个可选的存储参数。参数的详细描述如下所示：

  - FILLFACTOR

    一个表的填充因子（fillfactor）是一个介于10和100之间的百分数。100（完全填充）是默认值。如果指定了较小的填充因子，INSERT操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得UPDATE有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为100是优良选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数对于列存表没有意义。

    取值范围：10~100

  - ORIENTATION

    决定了表的数据的存储方式。

    取值范围：

    - COLUMN：表的数据将以列式存储。
    - ROW（缺省值）：表的数据将以行式存储。
    - ORC：表的数据将以ORC格式存储（仅HDFS表）。

    须知

    orientation不支持修改。

- COMPRESSION

  列存表的有效值为YES/NO和LOW/MIDDLE/HIGH，默认值为LOW。

  说明

  暂不支持行存表压缩功能。

- MAX_BATCHROW

  指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

  取值范围：10000~60000

  默认值：60000

- PARTIAL_CLUSTER_ROWS

  指定了在数据加载过程中进行将局部聚簇存储的记录数目。该参数只对列存表有效。

  取值范围：其有效值为大于等于10万。此值是MAX_BATCHROW的倍数。

- enable_delta

  指定了在列存表是否开启delta表。该参数只对列存表有效。

  默认值：off

- DELTAROW_THRESHOLD

  预留参数。该参数只对列存表有效。

  取值范围：0～60000，默认值为6000

- COLD_TABLESPACE

  指定冷分区保存的obs tablespace，仅冷热表支持。该参数仅支持列存分区表，且该参数不支持修改，需与storage_policy同时使用。在指定STORAGE_POLICY时，可不设置该参数，默认为default_obs_tbs。

  取值范围：有效的OBS TABLESPACE名。

- STORAGE_POLICY

  指定冷热分区切换规则，仅冷热表支持。该参数需与cold_tablespace同时使用。

  取值范围："冷热切换策略名称:冷热切换的阈值"，目前冷热切换的策略名称只支持LMT和HPN，LMT指按分区的最后更新时间切换，HPN指保留热分区的个数切换。

  - LMT：[day]：表示切换[day]时间前修改的热分区数据为冷分区，将该数据迁至OBS表空间中。其中[day]为整型，范围[0, 36500]，单位为天。
  - HPN: [hot_partition_num]：表示保留[hot_partition_num]个有数据的分区为热分区。保留规则为查找出有数据的分区的最大的Sequence ID，大于Sequence ID的无数据分区为热分区，并按这个Sequence ID从大到小保留[hot_partition_num]个分区为热分区；分区Sequence ID小于保留的最小热分区的Sequence ID的分区为冷分区，在冷热切换时，需要将数据迁移至OBS表空间中。其中[hot_partition_num]为整型，范围为[0,1600]。

须知

- 实时数仓（单机部署）暂不支持冷热分区切换功能。
- 对于LIST分区，建议谨慎使用HPN策略，否则可能出现新增分区不是热分区的情况。

- PERIOD

  指定分区管理中自动创建分区的周期，并开启自动创建分区功能。仅支持行存、列存范围分区表、时序表以及冷热表；分区键唯一并且类型仅支持TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE；不支持存在maxvalue分区；(nowTime - boundaryTime) / PERIOD需要小于分区个数上限，其中nowTime为当前时间，boundaryTime为现有分区中最早的分区边界时间；不支持在小型机、加速集群、单机集群上使用。

  取值范围：1 hour ~ 100 years

须知

- 在兼容Teradata或MySQL的数据库中，分区键类型为DATE时，PERIOD不能小于1 day。
- 建分区表时，如果设置了PERIOD，则可以只指定分区键不指定分区。建表时将创建两个默认分区，这两个默认分区的分区时间范围均为PERIOD。其中，第一个默认分区的边界时间是大于当前时间的第一个整时/整天/整周/整月/整年的时间，具体选择哪种整点时间取决于PERIOD的最大单位；第二个默认分区的边界时间是第一个分区边界时间加PERIOD。假设当前时间是2022-02-17 16:32:45，各种情况的第一个默认分区的分区边界选择如**表12-27**。

  有关默认分区的更多内容，请参见**示例5**。
- 实时数仓（单机部署）暂不支持自动创建分区功能。

表 12-27 分区边界选择

period	period最大单位	第一个默认分区的分区边界
1hour	Hour	2022-02-17 17:00:00
1day	Day	2022-02-18 00:00:00
1month	Month	2022-03-01 00:00:00
13month	Year	2023-01-01 00:00:00

- TTL

  指定分区管理中分区过期的时间，并开启自动删除分区功能。不支持单独设置，必须要提前或同时设置PERIOD，并且要大于或等于PERIOD。

  取值范围：1 hour ~ 100 years

说明

- PERIOD指明按照时间划分的周期对数据进行分区，分区的大小可能对查询性能有影响，同时每隔周期时间会创建一个新的周期大小的分区，具体做法是以period周期，自动调用**proc_add_partition**函数。TTL（Time To Live）指明该表的数据保存周期，超过TTL周期的数据将被清理，具体做法是以period周期，自动调用**proc_drop_partition**函数。PERIOD和TTL的值为Interval类型，例如：“1 hour”，“1 day”，“1 week”，“1 month” ，“1 year”，“1 month 2 day 3 hour”。
- 实时数仓（单机部署）暂不支持自动删除分区功能。

– COLVERSION

指定列存存储格式的版本，支持不同存储格式版本之间的切换，但分区表不支持存储格式版本切换。

取值范围：

1.0：列存表的每列以一个单独的文件进行存储，文件名以relfilenode.C1.0、relfilenode.C2.0、relfilenode.C3.0等命名。

2.0：列存表的每列合并存储在一个文件中，文件名以relfilenode.C1.0命名。

默认值：2.0

需注意，OBS冷热表仅支持colversion 2.0格式。

说明

在建列存表时选择COLVERSION=2.0，相比于1.0存储格式，在以下场景中性能有明显提升：

1. 创建列存宽表场景下，建表时间显著减少。
2. roach备份数据场景下，备份时间显著减少。
3. build、catch up耗时显著减少。
4. 占用磁盘空间大小显著减少。

– SKIP_FPI_HINT

顺序扫描过程中，若需要写FPW(full page writes)日志时，该参数控制是否跳过设置HintBits操作。

默认值：false

说明

设置SKIP_FPI_HINT=true时，在对某表执行checkpoint操作后，若对该表进行顺序扫描，将不再产生Xlog。适用于查询次数较少的中间表，有效减少Xlog的大小，提升查询性能。

- **COMPRESS / NOCOMPRESS**

创建一个新表时，需要在创建表语句中指定关键字COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字NOCOMPRESS则不对表进行压缩。

缺省值为NOCOMPRESS，即不对元组数据进行压缩。

- **DISTRIBUTE BY**

指定表如何在节点之间分布或者复制。

取值范围：

– REPLICATION：表的每一行存在所有数据节点(DN)中，即每个数据节点都有完整的表数据。

- ROUNDROBIN：表的每一行被轮番地发送给各个DN，因此数据会被均匀地分布在各个DN中。（ROUNDROBIN仅8.1.2及以上版本支持）
- HASH（column_name）：对指定的列进行Hash，通过映射，把数据分布到指定DN。

须知

- 当指定DISTRIBUTE BY HASH (column_name)参数时，创建主键和唯一索引必须包含“ column_name”列。
- 当被参照表指定DISTRIBUTE BY HASH (column_name)参数时，参照表的外键必须包含“ column_name”列。

默认值：由GUC参数default_distribution_mode控制。

- 当default_distribution_mode=roundrobin时，DISTRIBUTE BY的默认值按如下规则选取：
  i. 若建表时包含主键/唯一约束，则选取HASH分布，分布列为主键/唯一约束对应的列。
  ii. 若建表时不包含主键/唯一约束，则选取ROUNDROBIN分布。
- 当default_distribution_mode=hash时，DISTRIBUTE BY的默认值按如下规则选取：
  i. 若建表时包含主键/唯一约束，则选取HASH分布，分布列为主键/唯一约束对应的列。
  ii. 若建表时不包含主键/唯一约束，但存在数据类型支持作分布列的列，则选取HASH分布，分布列为第一个数据类型支持作分布列的列。
  iii. 若建表时不包含主键/唯一约束，也不存在数据类型支持作分布列的列，选取ROUNDROBIN分布。

以下数据类型支持作为分布列：

- INTEGER TYPES：TINYINT，SMALLINT，INT，BIGINT，NUMERIC/DECIMAL
- CHARACTER TYPES：CHAR，BPCHAR，VARCHAR，VARCHAR2，NVARCHAR2，TEXT
- DATE/TIME TYPES：DATE，TIME，TIMETZ，TIMESTAMP，TIMESTAMPTZ，INTERVAL，SMALLDATETIME

- **TO { GROUP groupname | NODE (nodename [, ...]) }**

  TO GROUP指定创建表所在的Node Group，目前不支持hdfs表使用。TO NODE主要供内部扩容工具使用，一般用户不应该使用。

- **PARTITION BY RANGE(partition_key)**

  指定范围分区策略语法，partition_key为分区键的名称。

  （1）对于从句是VALUES LESS THAN的语法格式：

须知

对于从句是VALUE LESS THAN的语法格式，范围分区策略的分区键最多支持4列，且分区键只能是列名。当存在多个分区键时，一个列名只能出现一次，且相邻的两个分区键要使用逗号隔开。

该情形下，分区键支持的数据类型为：SMALLINT、INTEGER、BIGINT、DECIMAL、NUMERIC、REAL、DOUBLE PRECISION、CHARACTER VARYING(n)、VARCHAR(n)、CHARACTER(n)、CHAR(n)、CHARACTER、CHAR、TEXT、NVARCHAR2、NAME、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

（2）对于从句是START END的语法格式：

须知

对于从句是START END的语法格式，范围分区策略的分区键仅支持1列。

该情形下，分区键支持的数据类型为：SMALLINT、INTEGER、BIGINT、DECIMAL、NUMERIC、REAL、DOUBLE PRECISION、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

- **PARTITION BY LIST (partition_key,[...])**

  指定列表分区策略语法，partition_key为分区键的名称。

须知

列表分区策略的分区键最多支持4列。

列表分区策略分区键支持的数据类型为：TINYINT、SMALLINT、INTEGER、BIGINT、NUMERIC/DECIMAL、TEXT、NVARCHAR2、VARCHAR(n)、CHAR、BPCHAR、TIME、TIME WITH TIMEZONE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、DATE、INTERVAL、SMALLDATETIME

- **partition_less_than_item**

```
PARTITION partition_name VALUES LESS THAN ( { partition_value | DEFAULT } )
```

  范围分区策略下分区（简称为范围分区）的定义语法。partition_name为范围分区的名称。partition_value为范围分区的上边界，取值依赖于partition_key的类型。MAXVALUE表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

须知

- 每个分区都需要指定一个上边界。
- 分区上边界的类型应当和分区键的类型一致。
- 分区列表是按照分区上边界升序排列的，值较小的分区位于值较大的分区之前。
- 如果分区键由多个字段组成，比较大小时，先比较第一个字段，当第一个字段相等时比较第二个字段，以此类推。

- **partition_start_end_item**

```
PARTITION partition_name {START (partition_value) END (partition_value) EVERY (interval_value)}
                         | {START (partition_value) END (partition_value|MAXVALUE)}
                         | {START(partition_value)}
                         | {END (partition_value| MAXVALUE)}
```

  使用起始值以及间隔值定义范围分区的语法，各参数含义如下：

- partition_name：范围分区的名称或名称前缀，除以下情形外（假定其中的partition_name是p1），均为分区的名称。
  - 若该定义是START+END+EVERY从句，则语义上定义的分区的名称依次为p1_1, p1_2, ...。例如对于定义“PARTITION p1 START(1) END(4) EVERY(1)”，则生成的分区是：[1, 2), [2, 3) 和 [3, 4)，名称依次为p1_1, p1_2和p1_3，即此处的p1是名称前缀。
  - 若该定义是第一个分区定义，且该定义有START值，则范围（MINVALUE, START）将自动作为第一个实际分区，其名称为p1_0，然后该定义语义描述的分区名称依次为p1_1, p1_2, ...。例如对于完整定义“PARTITION p1 START(1), PARTITION p2 START(2)”，则生成的分区是：(MINVALUE, 1), [1, 2) 和 [2, MAXVALUE)，其名称依次为p1_0, p1_1和p2，即此处p1是名称前缀，p2是分区名称。这里MINVALUE表示最小值。
- partition_value：范围分区的端点值（起始或终点），取值依赖于partition_key的类型，不可是MAXVALUE。
- interval_value：对[START，END) 表示的范围进行切分，interval_value是指定切分后每个分区的宽度，不可是MAXVALUE；如果（END-START）值不能整除以EVERY值，则仅最后一个分区的宽度小于EVERY值。
- MAXVALUE：表示最大值，它通常用于设置最后一个范围分区的上边界。

须知

1. 在创建分区表若第一个分区定义含START值，则范围（MINVALUE，START）将自动作为实际的第一个分区。
2. START END语法需要遵循以下限制：
   - 每个partition_start_end_item中的START值（如果有的话，下同）必须小于其END值；
   - 相邻的两个partition_start_end_item，第一个的END值必须等于第二个的START值；
   - 每个partition_start_end_item中的EVERY值必须是正向递增的，且必须小于（END-START）值；
   - 每个分区包含起始值，不包含终点值，即形如：[起始值，终点值)，起始值是MINVALUE时则不包含；
   - 一个partition_start_end_item创建的每个分区所属的TABLESPACE一样；
   - partition_name作为分区名称前缀时，其长度不要超过57字节，超过时自动截断；
   - 在创建、修改分区表时请注意分区表的分区总数不可超过最大限制（32767）；
3. 在创建分区表时START END与LESS THAN语法不可混合使用。
4. 即使创建分区表时使用START END语法，备份（gs_dump）出的SQL语句也是VALUES LESS THAN语法格式。

- list_partition_item

```
PARTITION partition_name VALUES ( { (partition_value) [, ... ] | DEFAULT } )
```

列表分区策略下分区（简称为列表分区）的定义语法。partition_name为分区的名称。partition_value为列表分区边界的一个枚举值，取值依赖于partition_key的类型。DEFAULT表示默认分区的边界。

须知

对于列表分区表，存在以下约定和约束：

- 边界值为DEFAULT的分区，称之为默认分区。
- 每个列表分区表只能有一个DEFAULT分区。
- 分区表的所有分区数不超过32767个，所有分区的边界值个数不大于32767个。
- 不管分区键的个数，DEFAULT分区的边界只能是一个DEFAULT。
- 如果分区键由多个字段组成，每个partition_value需要包含所有分区键的值，当分区键只有一列时，partition_value两侧的括号可以省略，参见**示例4：创建多个分区键的列表分区表**。
- 如果分区键由多个字段组成，比较大小时，先逐个字段比较大小，任何一个字段值不一样即可认为是不一样的键值。
- 边界中不同的partition_value值不能重复。
- 数据插入时，如果数据的分区键值能匹配任何非DEFAULT分区的边界，那么数据会写入对应的分区；否则数据会写入DEFAULT分区。

- **{ ENABLE | DISABLE } ROW MOVEMENT**

  行迁移开关。

  如果进行UPDATE操作时，更新了元组在分区键上的值，造成了该元组所在分区发生变化，就会根据该开关给出报错信息，或者进行元组在分区间的转移。

  取值范围：

  - ENABLE：行迁移开关打开。
  - DISABLE（缺省值）：行迁移开关关闭。

    说明

    分区表不显示指定则默认不开启ROW MOVEMENT，此时不允许跨分区更新。ENABLE ROW MOVEMENT开启则允许跨分区更新，但此时如果有SELECT FOR UPDATE查询该分区表并发执行，存在查询结果瞬时不一致的可能性，需要谨慎使用。

- **NOT NULL**

  字段值不允许为NULL。ENABLE用于语法兼容，可省略。

- **NULL**

  字段值允许NULL ，这是缺省。

  这个子句只是为和非标准SQL数据库兼容。不建议使用。

- **CHECK (condition) [NO INHERIT]**

  CHECK约束声明一个布尔表达式，每次要插入的新行或者要更新的行的新值必须使表达式结果为真或未知才能成功，否则会抛出一个异常并且不会修改数据库。

  声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以引用多个字段。

  用NO INHERIT标记的约束将不会传递到子表中去。

ENABLE用于语法兼容，可省略。

- **DEFAULT default_expr**

  DEFAULT子句给字段指定缺省值。该数值可以是任何不含变量的表达式(不允许使用子查询和对本表中的其他字段的交叉引用)。缺省表达式的数据类型必须和字段类型匹配。

  缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为NULL 。

- **UNIQUE index_parameters**

  UNIQUE (column_name [, ...]) index_parameters

  UNIQUE约束表示表里的一个字段或多个字段的组合必须在全表范围内唯一。

  对于唯一约束，NULL被认为是互不相等的。

  说明

  如果没有声明DISTRIBUTE BY REPLICATION，则唯一约束的列集合中必须包含分布列。

- **PRIMARY KEY index_parameters**

  PRIMARY KEY (column_name [, ...]) index_parameters

  主键约束声明表中的一个或者多个字段只能包含唯一的非NULL值。

  一个表只能声明一个主键。

  说明

  如果没有声明DISTRIBUTE BY REPLICATION，则主键约束的列集合中必须包含分布列。

- **DEFERRABLE | NOT DEFERRABLE**

  这两个关键字设置该约束是否可推迟。一个不可推迟的约束将在每条命令之后马上检查。可推迟约束可以推迟到事务结尾使用SET CONSTRAINTS命令检查。缺省是NOT DEFERRABLE。目前，行存的UNIQUE约束和主键约束可以接受这个子句。所有其他约束类型都是不可推迟的。

- **INITIALLY IMMEDIATE | INITIALLY DEFERRED**

  如果约束是可推迟的，则这个子句声明检查约束的缺省时间。

  - 如果约束是INITIALLY IMMEDIATE（缺省），则在每条语句执行之后就立即检查它；
  - 如果约束是INITIALLY DEFERRED ，则只有在事务结尾才检查它。

  约束检查的时间可以用SET CONSTRAINTS命令修改。

示例

- 示例1：使用LESS THAN语法创建range分区表。

  Range分区表customer_address含有4个分区，分区键为integer类型。分区的范围分别为：ca_address_sk<2450815，2450815<= ca_address_sk< 2451179，2451179<= ca_address_sk< 2451544，2451544<=ca_address_sk。

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
```

```
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
```

查看分区表信息：

```
SELECT relname, boundaries FROM pg_partition p where p.parentid='customer_address'::regclass
ORDER BY 1;
     relname      | boundaries
------------------+------------
 customer_address |
 p1               | {2450815}
 p2               | {2451179}
 p3               | {2451544}
 p4               | {NULL}
(5 rows)
```

查询分区P1的行数：

```
SELECT count(*) FROM customer_address PARTITION (P1);
SELECT count(*) FROM customer_address PARTITION FOR (2450815);
```

- 示例2：使用START END语法创建列存range分区表。

```
CREATE TABLE customer_address_SE
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
WITH (ORIENTATION = COLUMN)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
    PARTITION p1 START(1) END(1000) EVERY(200),
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(5000)
);
```

查看分区表信息：

```
SELECT relname, boundaries FROM pg_partition p where p.parentid='customer_address_SE'::regclass
ORDER BY 1;
      relname      | boundaries
-------------------+------------
 customer_address_se |
 p1_0               | {1}
 p1_1               | {201}
 p1_2               | {401}
 p1_3               | {601}
 p1_4               | {801}
 p1_5               | {1000}
 p2                 | {2000}
 p3                 | {5000}
(9 rows)
```

- 示例3：创建一个分区键的list分区表。

```
CREATE TABLE data_list
(
    id int,
    time int,
    sarlay decimal(12,2)
)
PARTITION BY LIST (time)
(
        PARTITION P1 VALUES (202209),
```



```
    PARTITION P2 VALUES (202210,202208),
    PARTITION P3 VALUES (202211),
    PARTITION P4 VALUES (202212),
    PARTITION P5 VALUES (202301)
);
```

- 示例4：创建多个分区键的list分区表。

  分区表有两个分区键：period, city。

```
CREATE TABLE sales_info
(
sale_time  timestamptz,
period     int,
city       text,
price      numeric(10,2),
remark     varchar2(100)
)
DISTRIBUTE BY HASH(sale_time)
PARTITION BY LIST (period, city)
(
PARTITION north_2022 VALUES (('202201', 'north1'), ('202202', 'north2')),
PARTITION south_2022 VALUES (('202201', 'south1'), ('202202', 'south2'), ('202203', 'south2')),
PARTITION rest VALUES (DEFAULT)
);
```

- 示例5：创建不指定分区的自动分区管理分区表，指定分区管理中自动创建分区的周期PERIOD为1 day，分区键为time。

```
CREATE TABLE time_part
 (
   id integer,
   time timestamp
 ) with (PERIOD='1 day')
 partition by range(time);
```

  建表时将创建两个默认分区，第一个默认分区的边界时间是大于当前时间的第一个整天的时间，即2022-12-13 00:00:00；第二个默认分区的边界时间是第一个分区边界时间加PERIOD，即2022-12-13 00:00:00+1day=2022-12-14 00:00:00。

```
SELECT now();
              now
-------------------------------
 2022-12-12 20:41:21.603172+08
(1 row)

SELECT relname, boundaries FROM pg_partition p where p.parentid='time_part'::regclass ORDER BY 1;
    relname     |       boundaries
----------------+-------------------------
 default_part_1 | {"2022-12-13 00:00:00"}
 default_part_2 | {"2022-12-14 00:00:00"}
 time_part      |
(3 rows)
```

- 示例6：创建指定分区的自动分区管理分区表。

```
CREATE TABLE CPU(
   id integer,
   idle numeric,
   IO numeric,
   scope text,
   IP text,
   time timestamp
) with (TTL='7 days',PERIOD='1 day')
partition by range(time)
(
   PARTITION P1 VALUES LESS THAN('2022-01-05 16:32:45'),
   PARTITION P2 VALUES LESS THAN('2022-01-06 16:56:12')
);
```

- 示例7：按照月份创建分区表customer_address，含有13个分区，分区键为date类型。

  创建分区表customer_address：


```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       integer           NOT NULL,
    ca_address_date       date            NOT NULL
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE (ca_address_date)
(
        PARTITION p202001 VALUES LESS THAN('20200101'),
        PARTITION p202002 VALUES LESS THAN('20200201'),
        PARTITION p202003 VALUES LESS THAN('20200301'),
        PARTITION p202004 VALUES LESS THAN('20200401'),
        PARTITION p202005 VALUES LESS THAN('20200501'),
        PARTITION p202006 VALUES LESS THAN('20200601'),
        PARTITION p202007 VALUES LESS THAN('20200701'),
        PARTITION p202008 VALUES LESS THAN('20200801'),
        PARTITION p202009 VALUES LESS THAN('20200901'),
        PARTITION p202010 VALUES LESS THAN('20201001'),
        PARTITION p202011 VALUES LESS THAN('20201101'),
        PARTITION p202012 VALUES LESS THAN('20201201'),
        PARTITION p202013 VALUES LESS THAN(MAXVALUE)
);
```

插入数据：

```
INSERT INTO customer_address values('1','20200215');
INSERT INTO customer_address values('7','20200805');
INSERT INTO customer_address values('9','20201111');
INSERT INTO customer_address values('4','20201231');
```

查询分区：

```
SELECT * FROM customer_address PARTITION(p202009);
 ca_address_sk |    ca_address_date
---------------+---------------------
             7 | 2020-08-05 00:00:00
(1 row)
```

- 示例8：使用START END语法一次创建含有多个分区的分区表。
  - 创建分区表day_part，每一天为一个分区，分区键为date类型。

    ```
    CREATE table day_part(id int,d_time date)
    DISTRIBUTE BY HASH (id)
    PARTITION BY RANGE (d_time)
    (PARTITION p1 START('2022-01-01') END('2022-01-31') EVERY(interval '1 day'));
    ALTER TABLE  day_part ADD PARTITION pmax VALUES LESS THAN (maxvalue);
    ```

  - 创建分区表week_part，每7天为一个分区，分区键为date类型。

    ```
    CREATE table week_part(id int,w_time date)
    DISTRIBUTE BY HASH (id)
    PARTITION BY RANGE (w_time)
    (PARTITION p1 START('2021-01-01') END('2022-01-01') EVERY(interval '7 day'));
    ALTER TABLE  week_part ADD PARTITION pmax VALUES LESS THAN (maxvalue);
    ```

  - 创建分区表month_part，每一个月为一个分区，分区键为date类型。

    ```
    CREATE table month_part(id int,m_time date)
    DISTRIBUTE BY HASH (id)
    PARTITION BY RANGE (m_time)
    (PARTITION p1 START('2021-01-01') END('2022-01-01') EVERY(interval '1 month'));
    ALTER TABLE  month_part ADD PARTITION pmax VALUES LESS THAN (maxvalue);
    ```

- 示例9：创建冷热表。

  仅支持列存分区表，使用obs默认表空间，冷热切换规则设置LMT为30。

  ```
  DROP TABLE IF EXISTS cold_hot_table;
  CREATE TABLE cold_hot_table
  (
      W_WAREHOUSE_ID          CHAR(16)             NOT NULL,
      W_WAREHOUSE_NAME          VARCHAR(20)                ,
      W_STREET_NUMBER         CHAR(10)                   ,
      W_STREET_NAME           VARCHAR(60)                ,
      W_STREET_ID             CHAR(15)                   ,
  ```



```
    W_SUITE_NUMBER          CHAR(10)
)
WITH (ORIENTATION = COLUMN, storage_policy = 'LMT:30')
DISTRIBUTE BY HASH (W_WAREHOUSE_ID)
PARTITION BY RANGE(W_STREET_ID)
(
    PARTITION P1 VALUES LESS THAN(100000),
    PARTITION P2 VALUES LESS THAN(200000),
    PARTITION P3 VALUES LESS THAN(300000),
    PARTITION P4 VALUES LESS THAN(MAXVALUE)
)ENABLE ROW MOVEMENT;
```

相关链接

ALTER TABLE PARTITION，**DROP TABLE**

12.53 CREATE TEXT SEARCH CONFIGURATION

功能描述

创建新的文本搜索配置。一个文本搜索配置声明一个能将一个字符串划分成符号的文本搜索解析器，加上可以用于确定搜索对哪些标记感兴趣的字典。

注意事项

- 若仅声明分析器，那么新的文本搜索配置初始没有从符号类型到词典的映射， 因此会忽略所有的单词。后面必须调用ALTER TEXT SEARCH CONFIGURATION命令创建映射使配置生效。如果声明了COPY选项，那么会自动复制指定的文本搜索配置的解析器、映射、配置选项等信息。
- 若模式名称已给出，那么文本搜索配置会在声明的模式中创建，否则会在当前模式创建。
- 定义文本搜索配置的用户成为其所有者。
- PARSER和COPY选项是互相排斥的，因为当一个现有配置被复制，其分析器配置也被复制了。

语法格式

```
CREATE TEXT SEARCH CONFIGURATION name
    ( PARSER = parser_name | COPY = source_config )
    [ WITH ( {configuration_option = value} [, ...] )];
```

参数说明

- **name**

  要创建的文本搜索配置的名称。该名称可以有模式修饰。

- **parser_name**

  用于该配置的文本搜索分析器的名称。

- **source_config**

  要复制的现有文本搜索配置的名称。

- **configuration_option**

  文本搜索配置的配置参数，主要是针对parser_name执行的解析器或者source_config隐含的解析器而言的。

取值范围：目前共支持default、ngram、zhparser三种类型的解析器，其中default类型的解析器没有对应的configuration_option，ngram、zhparser类型解析器对应的configuration_option如表12-28所示。

表 12-28 ngram、zhparser 类型解析器对应的配置参数

解析器	配置参数	参数描述	取值范围
ngram	gram_size	分词长度。	正整数，1~4 默认值：2
	punctuation_ignore	是否忽略标点符号。	• true（默认值）：忽略标点符号。 • false：不忽略标点符号。
	grapsymbol_ignore	是否忽略图形化字符。	• true：忽略图形化字符。 • false（默认值）：不忽略图形化字符。
zhparser	punctuation_ignore	分词结果是否忽略所有的标点等特殊符号（不会忽略\r和\n）。	• true（默认值）：忽略所有的标点等特殊符号。 • false：不忽略所有的标点等特殊符号。
	seg_with_duality	是否将闲散文字自动以二字分词法聚合。	• true：将闲散文字自动以二字分词法聚合。 • false（默认值）：不将闲散文字自动以二字分词法聚合。
	multi_short	分词执行时是否执行针对长词复合切分。	• true（默认值）：执行针对长词复合切分。 • false：不执行针对长词复合切分。
	multi_duality	设定是否将长词内的文字自动以二字分词法聚合。	• true：将长词内的文字自动以二字分词法聚合。 • false（默认值）：不将长词内的文字自动以二字分词法聚合。
	multi_zmain	是否将重要单字单独显示。	• true：将重要单字单独显示。 • false（默认值）：不将重要单字单独显示。

解析器	配置参数	参数描述	取值范围
	multi_zall	是否将全部单字单独显示。	• true：将全部单字单独显示。 • false（默认值）：不将全部单字单独显示。

示例

创建文本搜索配置：

```
DROP TEXT SEARCH CONFIGURATION IF EXISTS ngram1;
CREATE TEXT SEARCH CONFIGURATION ngram1 (parser=ngram) WITH (gram_size = 2, grapsymbol_ignore = false);
```

创建文本搜索配置：

```
DROP TEXT SEARCH CONFIGURATION IF EXISTS ngram2;
CREATE TEXT SEARCH CONFIGURATION ngram2 (copy=ngram1) WITH (gram_size = 2, grapsymbol_ignore = false);
```

创建文本搜索配置：

```
DROP TEXT SEARCH CONFIGURATION IF EXISTS english_1;
CREATE TEXT SEARCH CONFIGURATION english_1 (parser=default);
```

相关链接

ALTER TEXT SEARCH CONFIGURATION，**DROP TEXT SEARCH CONFIGURATION**

12.54 CREATE TEXT SEARCH DICTIONARY

功能描述

创建一个新的全文检索词典。词典是一种指定在全文检索时识别特定词并处理的方法。

词典的创建依赖于预定义模板（在系统表PG_TS_TEMPLATE中定义），支持创建五种类型的词典，分别是Simple、Ispell、Synonym、Thesaurus以及Snowball，每种类型的词典可以完成不同的任务。

注意事项

- 具有SYSADMIN权限的用户可以执行创建词典操作，创建该词典的用户自动成为其所有者。
- 临时模式（pg_temp）下不允许创建词典。
- 创建或修改词典之后，任何对于用户自定义的词典定义文件的修改，将不会影响到数据库中的词典。如果需要在数据库中使用这些修改，需使用ALTER语句更新对应词典的定义文件。

语法格式

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
);
```

参数说明

- **name**

  要创建的词典的名称（可指定模式名，否则在当前模式下创建）。

  取值范围：符合标识符命名规范的字符串，且最大长度不超过63个字符。

- **template**

  模板名。

  取值范围：系统表PG_TS_TEMPLATE中定义的模板：Simple/Synonym/Thesaurus/Ispell/Snowball。

- **option**

  参数名。与template值对应，不同的词典模板具有不同的参数列表，且与指定顺序无关。

  - Simple词典对应的option

    - **STOPWORDS**

      停用词表文件名，默认后缀名为stop。例如STOPWORDS = french，则实际文件名为french.stop。停用词文件格式为一组word列表，每行定义一个停用词。词典处理时，文件中的空行和空格会被忽略，并将stopword词组转换为小写形式。

    - **ACCEPT**

      是否将非停用词设置为已识别。默认值为true。

      当Simple词典设置参数ACCEPT=true时，将不会传递任何token给后继词典，此时建议将其放置在词典列表的最后。反之，当ACCEPT=false时，建议将该Simple词典放置在列表中的至少一个词典之前。

    - **FILEPATH**

      停用词文件所在目录。停用词文件可以存放于本地，也可以存放于对象存储服务OBS服务器。如果存在本地，目录格式为'file://absolute_path'。如果存放于OBS服务器，目录格式为'obs://bucket/path accesskey=ak secretkey=sk region=region_name'。注意目录要用单引号包含。默认值为预定义词典文件所在目录。FILEPATH参数必须和STOPWORDS参数同时指定，不允许单独指定。

      使用OBS服务器上的停用词文件创建字典的过程如下：

      1) 将停用词文件上传到OBS服务器。例如将french.stop文件上传到OBS服务器obsv3.sa-fb-1.externaldemo.com上名为gaussdb的桶中，其URL为https://gaussdb.obsv3.sa-fb-1.externaldemo.com/french.stop。上传文件及查询URL的方式请参考OBS用户手册。
      2) 修改$GAUSSHOME/etc/region_map文件，在其中加入一行"region_name": "obs domain"。region_name可以为任意由大小写字母、数字、斜杠（/）或下划线组成的字符串。obs domain为OBS服务器的域名。

         示例，将region_name设为rg，region_map包含的内容如："rg": "obsv3.sa-fb-1.externaldemo.com"。

须知

region_name和obs domain都用双引号，冒号的左边没有空格，右边有1个空格。

3) 执行CREATE TEXT SEARCH DICTIONARY 命令创建字典。命令如下：

```
CREATE TEXT SEARCH DICTIONARY french_dict ( TEMPLATE = pg_catalog.simple, STOPWORDS = french, FILEPATH = 'obs://gaussdb accesskey=xxx secretkey=yyy region=rg' );
```

由于french.stop文件放在gaussdb桶的根目录下，因此path为空。

- Synonym词典对应的option
  - **SYNONYM**

    同义词词典的定义文件名，默认后缀名为syn。

    文件格式为一组同义词列表，每行格式为"token synonym"，即token和其对应的synonym，中间以空格相连。

  - **CASESENSITIVE**

    设置是否大小写敏感，默认值为false，此时词典文件中的token和synonym均会转为小写形式处理。如果设置为true，则不会进行小写转换。

  - **FILEPATH**

    同义词词典文件所在目录。目录可以指定为本地目录和OBS目录两种形式。默认值为预定义词典文件所在目录。其中目录格式、以及使用OBS服务器上的文件创建Synonym字典的过程与**Simple词典的FILEPATH**相同。

- Thesaurus词典对应的option
  - **DICTFILE**

    词典定义文件名，默认后缀名为ths。

    文件格式为一组同义词列表，每行格式为"sample words : indexed words"，中间冒号（:）作为短语和其替换词间的分隔符。TZ词典处理时，如果有多个匹配的sample words，将选择最长匹配输出。

  - **DICTIONARY**

    用于词规范化的子词典名，必须且仅能定义一个。该词典必须是已经存在的，在检查短语匹配之前使用，用于识别和规范输入文本。

    如果子词典无法识别输入词，将会报错。此时，需要移除该词或者更新子词典使其识别。此外，可在indexed words的开头放上一个星号（*）来跳过在其上应用子词典，但是所有sample words必须可以被子词典识别。

    如果词典文件定义的sample words中，含有子词典中定义的停用词，需要用问号（？）替代停用词。假设a和the是子词典中所定义的停用词，如下：

    ```
    ? one ? two : swsw
    ```

    上述同义词组定义会匹配"a one the two"以及"the one a two"，这两个短语均会被swsw替代输出。

- **FILEPATH**

  词典定义文件所在目录。目录可以指定为本地目录和OBS目录两种形式。默认值为预定义词典文件所在目录。其中目录格式、以及使用OBS服务器上的文件创建Synonym字典的过程与**Simple词典的FILEPATH**相同。

– Ispell词典

- **DICTFILE**

  词典定义文件名，默认后缀名为dict。

- **AFFFILE**

  词缀文件名，默认后缀名为affix。

- **STOPWORDS**

  停用词文件名，默认后缀名为stop，文件格式要求与Simple类型词典的停用词文件相同。

- **FILEPATH**

  词典文件所在目录。可以指定为本地目录和OBS目录两种形式。默认值为预定义词典文件所在目录。其中目录格式、以及使用OBS服务器上的文件创建Synonym字典的过程与**Simple词典的FILEPATH**相同。

– Snowball词典

- **LANGUAGE**

  语言名，标识使用哪种语言的词干分析算法。算法按照对应语言中的拼写规则，缩减输入词的常见变体形式为一个基础词或词干。

- **STOPWORDS**

  停用词表文件名，默认后缀名为stop，文件格式要求与Simple类型词典的停用词文件相同。

- **FILEPATH**

  词典定义文件所在目录。可以指定为本地目录或者OBS目录。默认值为预定义词典文件所在目录。FILEPATH参数必须和STOPWORDS参数同时指定，不允许单独指定。其中目录格式、以及用OBS服务器上的文件创建Snowball字典的过程与Simple字典相同。

说明

- 预定义词典文件位于$GAUSSHOME/share/postgresql/tsearch_data目录下。
- 词典定义文件的文件名仅支持小写字母、数字、下划线混合。

- **value**

  参数值。如果不是简单的标识符或数字，则参数值必须加单引号（标示符和数字同样可以加上单引号）。

示例

创建一个Ispell词典english_ispell（词典定义文件来自开源词典）：

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
DROP TEXT SEARCH DICTIONARY IF EXISTS english_ispell;
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english,
    FilePath = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'
);
```

创建一个Snowball词典english_snowball（词典定义文件来自开源词典）：

须知

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
DROP TEXT SEARCH DICTIONARY IF EXISTS english_snowball;
CREATE TEXT SEARCH DICTIONARY english_snowball (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english,
    FilePath = 'obs://bucket_name/path accesskey=ak secretkey=sk region=rg'
);
```

相关链接

ALTER TEXT SEARCH DICTIONARY，**CREATE TEXT SEARCH DICTIONARY**

12.55 CREATE TRIGGER

功能描述

创建一个触发器。 触发器将与指定的表或视图关联，并在特定条件下执行指定的函数。

注意事项

- 当前仅支持在普通行存表上创建触发器，不支持在列存表、临时表、unlogged表等类型表上创建触发器。
- 如果为同一事件定义了多个相同类型的触发器，则按触发器的名称字母顺序触发它们。
- 一个触发器只能作用在一张表上，对创建的触发器数量无限制但一个表上的触发器越多，性能消耗越大。
- 触发器常用于多表间数据关联同步场景，对SQL执行性能影响较大，不建议在大数据量同步及对性能要求高的场景中使用。
- 当触发器满足如下条件时，触发语句能和触发器一起下推到DN执行并提升触发器执行性能：

- 开关enable_trigger_shipping和enable_fast_query_shipping开启（默认均开启）。
- 源表触发器使用的触发器函数为plpgsql类型（推荐类型）。
- 源表与触发表分布键的类型、数量完全相同，均为行存表，且所属的nodegroup相同。
- 原INSERT/UPDATE/DELETE语句条件中包含所有分布键与NEW/OLD等值比较表达式。
- 原INSERT/UPDATE/DELETE语句在没有触发器的情况下原本就能query shipping。
- 源表上只有INSERT/UPDATE/DELETE AFTER/BEFORE FOR EACH ROW六类触发器，且所有触发器都可下推。

语法格式

```
CREATE [ CONSTRAINT ] TRIGGER trigger_name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
   ON table_name
   [ FROM referenced_table_name ]
   { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }
   [ FOR [ EACH ] { ROW | STATEMENT } ]
   [ WHEN ( condition ) ]
   EXECUTE PROCEDURE function_name ( arguments );
```

其中event包含以下几种：

```
   INSERT
   UPDATE [ OF column_name [, ... ] ]
   DELETE
   TRUNCATE
```

参数说明

- **CONSTRAINT**

  可选项，指定此参数将创建约束触发器，即触发器作为约束来使用。除了可以使用SET CONSTRAINTS调整触发器触发的时间之外，这与常规触发器相同。 约束触发器必须是AFTER ROW触发器。

- **trigger_name**

  触发器名称，该名称不能限定模式，因为触发器自动继承其所在表的模式，且同一个表的触发器不能重名。 对于约束触发器，使用**SET CONSTRAINTS**修改触发器行为时也使用此名称。

  取值范围：符合标识符命名规范的字符串，且最大长度不超过63个字符。

- **BEFORE**

  触发器函数是在触发事件发生前执行。

- **AFTER**

  触发器函数是在触发事件发生后执行，约束触发器只能指定为AFTER。

- **INSTEAD OF**

  触发器函数直接替代触发事件。

- **event**

  启动触发器的事件，取值范围包括：INSERT、UPDATE、DELETE或TRUNCATE，也可以通过OR同时指定多个触发事件。

  对于UPDATE事件类型，可以使用下面语法指定列：

  ```
  UPDATE OF column_name1 [, column_name2 ... ]
  ```

表示只有这些列作为UPDATE语句的目标列时，才会启动触发器，但是INSTEAD OF UPDATE类型不支持指定列信息。

- **table_name**

  需要创建触发器的表名称。

  取值范围：数据库中已经存在的表名称。

- **referenced_table_name**

  约束引用的另一个表的名称。 只能为约束触发器指定，常见于外键约束。由于当前不支持外键，因此不建议使用。

  取值范围：数据库中已经存在的表名称。

- **DEFERRABLE | NOT DEFERRABLE**

  约束触发器的启动时机，仅作用于约束触发器。这两个关键字设置该约束是否可推迟。

  详细介绍请参见**CREATE TABLE**。

- **INITIALLY IMMEDIATE | INITIALLY DEFERRED**

  如果约束是可推迟的，则这个子句声明检查约束的缺省时间，仅作用于约束触发器。

  详细介绍请参见**CREATE TABLE**。

- **FOR EACH ROW | FOR EACH STATEMENT**

  触发器的触发频率。

  - FOR EACH ROW是指该触发器是受触发事件影响的每一行触发一次。
  - FOR EACH STATEMENT是指该触发器是每个SQL语句只触发一次。

  未指定时默认值为FOR EACH STATEMENT。约束触发器只能指定为FOR EACH ROW。

- **condition**

  决定是否实际执行触发器函数的条件表达式。当指定WHEN时，只有在条件返回true时才会调用该函数。

  在FOR EACH ROW触发器中，WHEN条件可以通过分别写入OLD.column_name或NEW.column_name来引用旧行或新行值的列。 需要注意，INSERT触发器不能引用OLD和DELETE触发器不能引用NEW。

  INSTEAD OF触发器不支持WHEN条件。

  WHEN表达式不能包含子查询。

  对于约束触发器，WHEN条件的评估不会延迟，而是在执行更新操作后立即发生。 如果条件返回值不为true，则触发器不会排队等待延迟执行。

- **function_name**

  用户定义的函数，必须声明为不带参数并返回类型为触发器，在触发器触发时执行。

- **arguments**

  执行触发器时要提供给函数的可选的以逗号分隔的参数列表。参数是文字字符串常量，简单的名称和数字常量也可以写在这里，但它们都将被转换为字符串。 请检查触发器函数的实现语言的描述，以了解如何在函数内访问这些参数。

说明

关于触发器种类：

- INSTEAD OF的触发器必须标记为FOR EACH ROW，并且只能在视图上定义。
- BEFORE和AFTER触发器作用在视图上时，只能标记为FOR EACH STATEMENT。
- TRUNCATE类型触发器仅限FOR EACH STATEMENT。

表 12-29 表和视图上支持的触发器种类：

触发时机	触发事件	行级	语句级
BEFORE	INSERT/UPDATE/DELETE	表	表和视图
	TRUNCATE	不支持	表
AFTER	INSERT/UPDATE/DELETE	表	表和视图
	TRUNCATE	不支持	表
INSTEAD OF	INSERT/UPDATE/DELETE	视图	不支持
	TRUNCATE	不支持	不支持

表 12-30 PLPGSQL 类型触发器函数特殊变量：

变量名	变量含义
NEW	INSERT及UPDATE操作涉及tuple信息中的新值，对DELETE为空。
OLD	UPDATE及DELETE操作涉及tuple信息中的旧值，对INSERT为空。
TG_NAME	触发器名称。
TG_WHEN	触发器触发时机（BEFORE/AFTER/INSTEAD OF）。
TG_LEVEL	触发频率（ROW/STATEMENT）。
TG_OP	触发操作（INSERT/UPDATE/DELETE/TRUNCATE）。
TG_RELID	触发器所在表OID。
TG_RELNAME	触发器所在表名（已废弃，现用TG_TABLE_NAME替代）。
TG_TABLE_NAME	触发器所在表名。
TG_TABLE_SCHEMA	触发器所在表的SCHEMA信息。

变量名	变量含义
TG_NARGS	触发器函数参数个数。
TG_ARGV[]	触发器函数参数列表。

示例

创建源表及触发表：

```
DROP TABLE IF EXISTS test_trigger_src_tbl;
DROP TABLE IF EXISTS test_trigger_des_tbl;

CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);
CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);
```

创建触发器函数tri_insert_func()：

```
DROP FUNCTION IF EXISTS tri_insert_func;
CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
        $$
        DECLARE
        BEGIN
                INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2, NEW.id3);
                RETURN NEW;
        END
        $$ LANGUAGE PLPGSQL;
```

创建触发器函数tri_update_func()：

```
DROP FUNCTION IF EXISTS tri_update_func;
CREATE OR REPLACE FUNCTION tri_update_func() RETURNS TRIGGER AS
        $$
        DECLARE
        BEGIN
                UPDATE test_trigger_des_tbl SET id3 = NEW.id3 WHERE id1=OLD.id1;
                RETURN OLD;
        END
        $$ LANGUAGE PLPGSQL;
```

创建触发器函数tri_delete_func()：

```
DROP FUNCTION IF EXISTS tri_delete_func;
CREATE OR REPLACE FUNCTION tri_delete_func() RETURNS TRIGGER AS
        $$
        DECLARE
        BEGIN
                DELETE FROM test_trigger_des_tbl WHERE id1=OLD.id1;
                RETURN OLD;
        END
        $$ LANGUAGE PLPGSQL;
```

创建INSERT触发器：

```
DROP FUNCTION IF EXISTS insert_trigger;
CREATE TRIGGER insert_trigger
        BEFORE INSERT ON test_trigger_src_tbl
        FOR EACH ROW
        EXECUTE PROCEDURE tri_insert_func();
```

创建UPDATE触发器：

```
DROP FUNCTION IF EXISTS update_trigger;
CREATE TRIGGER update_trigger
        AFTER UPDATE ON test_trigger_src_tbl
```

```
    FOR EACH ROW
    EXECUTE PROCEDURE tri_update_func();
```

创建DELETE触发器：

```
DROP FUNCTION IF EXISTS update_trigger;
CREATE TRIGGER delete_trigger
    BEFORE DELETE ON test_trigger_src_tbl
    FOR EACH ROW
    EXECUTE PROCEDURE tri_delete_func();
```

相关链接

ALTER TRIGGER，**DROP TRIGGER**，**ALTER TABLE**

12.56 CREATE TYPE

功能描述

在当前数据库中定义一种新的数据类型。定义数据类型的用户将成为该数据类型的拥有者。类型只适用于行存表。

有四种形式的CREATE TYPE，分别为：复合类型、基本类型、shell类型和枚举类型。

- 复合类型

  复合类型由一个属性名和数据类型的列表指定。如果属性的数据类型是可排序的，也可以指定该属性的排序规则。复合类型本质上和表的行类型相同，但是如果只想定义一种类型，使用CREATE TYPE避免了创建一个实际的表。单独的复合类型也是很有用的，例如可以作为函数的参数或者返回类型。

  为了能够创建复合类型，必须拥有在其所有属性类型上的USAGE特权。

- 基本类型

  用户可以自定义一种新的基本类型（标量类型）。通常来说这些函数必须是用C或者另外一种低层语言所编写。

- shell类型

  shell类型是一种用于后面要定义的类型的占位符，通过发出一个只带类型名参数的CREATE TYPE命令来创建该类型。在创建基本类型时，需要shell类型作为一种向前引用。

- 枚举类型

  由若干个标签构成的列表，每一个标签值都是一个非空字符串，且字符串长度必须不超过64个字节。

注意事项

如果给定一个模式名，那么该类型将被创建在指定的模式中，否则它会被创建在当前模式中。类型名称必须与同一个模式中任何现有的类型或者域有所区别（因为表具有相关的数据类型，类型名称也必须与同一个模式中任何现有表的名字不同）。

语法格式

```
CREATE TYPE name AS
   ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name (
```

```
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )
```

参数说明

复合类型

- **name**

  要创建的类型的名称（可以被模式限定）。

- **attribute_name**

  复合类型的一个属性（列）的名称。

- **data_type**

  要成为复合类型的一个列的现有数据类型的名称。

- **collation**

  要关联到复合类型的一列的现有排序规则的名称。

基本类型

自定义基本类型时，参数可以以任意顺序出现，input_function和output_function为必选参数，其它为可选参数。

- **input_function**

  将数据从类型的外部文本形式转换为内部形式的函数名。

  输入函数可以被声明为有一个cstring类型的参数，或者有三个类型分别为cstring、 oid、integer的参数。

  - cstring参数是以C字符串存在的输入文本。
  - oid参数是该类型自身的OID（对于数组类型则是其元素类型的OID）。
  - integer参数是目标列的typmod（如果知道，不知道则将传递 -1）。

  输入函数必须返回一个该数据类型本身的值。通常，一个输入函数应该被声明为STRICT。 如果不是这样，在读到一个NULL输入值时，调用输入函数时第一个参数会是NULL。在这种情况下，该函数必须仍然返回NULL，除非调用函数发生了错误（这种情况主要是想支持域输入函数，域输入函数可能需要拒绝NULL输入）。

说明

输入和输出函数能被声明为具有新类型的结果或参数是因为：必须在创建新类型之前创建这两个函数。而新类型应该首先被定义为一种shell type，它是一种占位符类型，除了名称和拥有者之外它没有其他属性。这可以通过不带额外参数的命令CREATE TYPE name做到。然后用C写的I/O函数可以被定义为引用这种shell type。最后，用带有完整定义的CREATE TYPE把该shell type替换为一个完全的、合法的类型定义，之后新类型就可以正常使用了。

- **output_function**

  将数据从类型的内部形式转换为外部文本形式的函数名。

  输出函数必须被声明为有一个新数据类型的参数。输出函数必须返回类型cstring。对于NULL值不会调用输出函数。

- **receive_function**

  可选参数。将数据从类型的外部二进制形式转换成内部形式的函数名。

  如果没有该函数，该类型不能参与到二进制输入中。二进制表达转换成内部形式代价更低，然而却更容易移植（例如，标准的整数数据类型使用网络字节序作为外部二进制表达，而内部表达是机器本地的字节序）。receive_function应该执行足够的检查以确保该值是有效的。

  接收函数可以被声明为有一个internal类型的参数，或者有三个类型分别为internal、oid、integer的参数。

  - internal参数是一个指向StringInfo缓冲区的指针，其中保存着接收到的字节串。
  - oid和integer参数和文本输入函数的相同。

  接收函数必须返回一个该数据类型本身的值。通常，一个接收函数应该被声明为STRICT。如果不是这样，在读到一个NULL输入值时调用接收函数时第一个参数会是NULL。在这种情况下，该函数必须仍然返回NULL，除非接收函数发生了错误（这种情况主要是想支持域接收函数，域接收函数可能需要拒绝NULL输入）。

- **send_function**

  可选参数。将数据从类型的内部形式转换为外部二进制形式的函数名。

  如果没有该函数，该类型将不能参与到二进制输出中。发送函数必须被声明为有一个新数据类型的参数。发送函数必须返回类型bytea。对于NULL值不会调用发送函数。

- **type_modifier_input_function**

  可选参数。将类型的修饰符数组转换为内部形式的函数名。

- **type_modifier_output_function**

  可选参数。将类型的修饰符的内部形式转换为外部文本形式的函数名。

  说明

  如果该类型支持修饰符（附加在类型声明上的可选约束，例如char(5)或numeric(30,2)），则需要可选的type_modifier_input_function以及type_modifier_output_function。GaussDB(DWS)允许用户定义的类型有一个或者多个简单常量或者标识符作为修饰符。不过，为了存储在系统目录中，该信息必须能被打包到一个非负整数值中。所声明的修饰符会被以cstring数组的形式传递给type_modifier_input_function。type_modifier_input_function必须检查该值的合法性（如果值错误就抛出一个错误），如果值正确，要返回一个非负integer值，该值将被存储在“typmod”列中。如果类型没有 type_modifier_input_function则类型修饰符将被拒绝。type_modifier_output_function把内部的整数typmod值转换回正确的形式用于用户显示。type_modifier_output_function必须返回一个cstring值，该值就是追加到类型名称后的字符串。例如，numeric的函数可能会返回(30,2)。如果默认的显示格式就是只把存储的typmod整数值放在圆括号内，则允许省略type_modifier_output_function。

- **analyze_function**

  可选参数。为该数据类型执行统计分析的函数名的可选参数。

  默认情况下，如果该类型有一个默认的B-tree操作符类，ANALYZE将尝试用类型的“equals”和“less-than”操作符来收集统计信息。这种行为对于非标量类型并不合适，因此可以通过指定一个自定义分析函数来覆盖这种行为。分析函数必须被声明为有一个类型为internal的参数，并且返回一个boolean结果。

- **internallength**

  可选参数。一个数字常量，用于指定新类型的内部表达的字节长度。默认为变长。

  虽然只有I/O函数和其他为该类型创建的函数才知道新类型的内部表达的细节， 但是内部表达的一些属性必须被向GaussDB(DWS)声明。其中最重要的是internallength。基本数据类型可以是定长的（这种情况下internallength是一个正整数）或者是变长的（把internallength设置为VARIABLE，在内部通过把typlen设置为-1表示）。所有变长类型的内部表达都必须以一个4字节整数开始，internallength定义了总长度。

- **PASSEDBYVALUE**

  可选参数。表示这种数据类型的值需要被传值而不是传引用。传值的类型必须是定长的，并且它们的内部表达不能超过Datum类型（某些机器上是4字节，其他机器上是8字节）的尺寸。

- **alignment**

  可选参数。该参数指定数据类型的存储对齐需求。如果被指定，必须是char、int2、int4或者double。默认是int4。

  允许的值等同于以1、2、4或8字节边界对齐。要注意变长类型的alignment参数必须至少为4，因为它们需要包含一个int4作为它们的第一个组成部分。

- **storage**

  可选参数。该数据类型的存储策略。

  如果被指定，必须是plain、external、extended或者main。 默认是plain。

  - plain指定该类型的数据将总是被存储在线内并且不会被压缩。（对定长类型只允许plain）
  - extended指定系统将首先尝试压缩一个长的数据值，并且将在数据仍然太长的情况下把值移出主表行。
  - external允许值被移出主表， 但是系统将不会尝试对它进行压缩。
  - main允许压缩，但是不鼓励把值移出主表（如果没有其他办法让行的大小变得合适，具有这种存储策略的数据项仍将被移出主表，但比起extended以及external项来，这种存储策略的数据项会被优先考虑保留在主表中）。

    除plain之外所有的storage值都暗示该数据类型的函数能处理被TOAST过的值。指定的值仅仅是决定一种可TOAST数据类型的列的默认TOAST存储策略，用户可以使用ALTER TABLE SET STORAGE为列选取其他策略。

- **like_type**

  可选参数。与新类型具有相同表达的现有数据类型的名称。会从这个类型中复制internallength、 passedbyvalue、 alignment以及storage的值（ 除非在这个CREATE TYPE命令的其他地方用显式说明覆盖）。

  当新类型的低层实现是以一种现有的类型为参考时，用这种方式指定表达特别有用。

- **category**

可选参数。这种类型的分类码（一个ASCII 字符）。 默认是“用户定义类型”的'U'。为了创建自定义分类， 也可以选择其他ASCII字符。

- **preferred**

  可选参数。如果这种类型是其类型分类中的优先类型则为TRUE，否则为FALSE。默认为FALSE。在一个现有类型分类中创建一种新的优先类型要非常谨慎， 因为这可能会导致很大的改变。

  说明

  category和preferred参数可以被用来帮助控制在混淆的情况下应用哪一种隐式造型。每一种数据类型都属于一个用单个ASCII 字符命名的分类，并且每一种类型可以是其所属分类中的“首选”。当有助于解决重载函数或操作符时，解析器将优先造型到首选类型（但是只能从同类的其他类型造型）。对于没有隐式转换到或来自任意其他类型的类型，让这些设置保持默认即可。不过，对于有隐式转换的相关类型的组，把它们都标记为属于同一个类别并且选择一种或两种“最常用”的类型作为该类别的首选通常是很有用的。在把一种用户定义的类型增加到一个现有的内建类别（例如，数字或者字符串类型）中时，category参数特别有用。不过，也可以创建新的全部是用户定义类型的类别。对这样的类别，可选择除大写字母之外的任何ASCII字符。

- **default**

  可选参数。数据类型的默认值。如果被省略，默认值是空。

  如果用户希望该数据类型的列被默认为某种非空值，可以指定一个默认值。默认值可以用DEFAULT关键词指定（这样一个默认值可以被附加到一个特定列的显式DEFAULT子句覆盖）。

- **element**

  可选参数。被创建的类型是一个数组，element指定了数组元素的类型。例如，要定义一个4字节整数的数组（int4）， 应指定ELEMENT = int4。

- **delimiter**

  可选参数。指定这种类型组成的数组中分隔值的定界符。

  可以把delimiter设置为一个特定字符，默认的定界符是逗号（,）。注意定界符是与数组元素类型相关的，而不是数组类型本身相关。

- **collatable**

  可选参数。如果这个类型的操作可以使用排序规则信息，则为TRUE。默认为FALSE。

  如果collatable为TRUE，这种类型的列定义和表达式可能通过使用COLLATE子句携带有排序规则信息。在该类型上操作的函数的实现负责真正利用这些信息，仅把类型标记为可排序的并不会让它们自动地去使用这类信息。

- **lable**

  可选参数。与枚举类型的一个值相关的文本标签，其值为长度不超过64个字符的非空字符串。

  说明

  在创建用户定义类型的时候， GaussDB(DWS)会自动创建一个与之关联的数组类型，其名字由该元素类型的名字前缀一个下划线组成。

示例

示例一：创建一种复合类型，建表并插入数据以及查询。

```
CREATE TYPE compfoo AS (f1 int, f2 text);
CREATE TABLE t1_compfoo(a int, b compfoo);
CREATE TABLE t2_compfoo(a int, b compfoo);
```



```
INSERT INTO t1_compfoo values(1,(1,'demo'));
INSERT INTO t2_compfoo select * from t1_compfoo;
SELECT (b).f1 FROM t1_compfoo;
SELECT * FROM t1_compfoo t1 join t2_compfoo t2 on (t1.b).f1=(t1.b).f1;
```

示例二：创建一个枚举类型，并在表定义中使用它。

```
CREATE TYPE bugstatus AS ENUM ('create', 'modify', 'closed');
CREATE TABLE customer (name text,current_bugstatus bugstatus);
INSERT INTO customer VALUES ('type','create');
SELECT * FROM customer WHERE current_bugstatus = 'create';
```

示例三：编译.so文件，并创建shell类型。

```
CREATE TYPE complex;
```

这个语句的作用是为要定义的类型创建了一个占位符，这样允许在定义其I/O函数时引用该类型。现在可以定义 I/O函数，需要注意的是在创建函数时function必须声明为NOT FENCED模式：

```
CREATE FUNCTION
complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_recv(internal)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT not fenced;

CREATE FUNCTION
complex_send(complex)
    RETURNS bytea
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT not fenced;
```

--最后，提供该数据类型的完整定义：

```
CREATE TYPE complex (
internallength = 16,
input = complex_in,
output = complex_out,
receive = complex_recv,
send = complex_send,
alignment = double
);
```

input、output、receive及send函数对应的C函数定义如下：

```
--定义结构体Complex如下：
typedef struct Complex {
    double      x;
    double      y;
} Complex;

--定义input函数：
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
```



```
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}

--定义output函数：
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
        Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
        char       *result;

        result = (char *) palloc(100);
        snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
        PG_RETURN_CSTRING(result);
}

--定义receive函数：
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

--定义send函数：
PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

相关链接

ALTER TYPE，**DROP TYPE**

12.57 CREATE USER

功能描述

创建一个用户。

注意事项

- 通过CREATE USER创建的用户，默认具有LOGIN权限；
- 通过CREATE USER创建用户的同时系统会在执行该命令的数据库中，为该用户创建一个同名的SCHEMA；其他数据库中，则不自动创建同名的SCHEMA；用户可使用CREATE SCHEMA命令，分别在其他数据库中，为该用户创建同名SCHEMA。
- 系统管理员在普通用户同名schema下创建的对象，所有者为schema的同名用户（非系统管理员）。
- 除系统管理员之外，其他用户即使被授权了schema的所有权限也无法在普通用户的同名schema下创建对象，除非把同名schema相关的角色权限赋予其他用户。具体操作可参考“**赋予用户schema的all权限后建表仍然报错**”章节。

语法格式

```
CREATE USER user_name [ [ WITH ] option [ ... ] ] [ ENCRYPTED | UNENCRYPTED ] { PASSWORD |
IDENTIFIED BY } { 'password' | DISABLE };
```

其中option子句用于设置权限及属性等信息。

```
{SYSADMIN | NOSYSADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {INDEPENDENT | NOINDEPENDENT}
    | {VCADMIN | NOVCADMIN}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | RESOURCE POOL 'respool'
    | USER GROUP 'groupuser'
    | PERM SPACE 'spacelimit'
    | TEMP SPACE 'tmpspacelimit'
    | SPILL SPACE 'spillspacelimit'
    | NODE GROUP logic_cluster_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
    | AUTHINFO 'authinfo'
    | PASSWORD EXPIRATION period
```

参数说明

- **user_name**

  用户名称。

  取值范围：字符串，要符合标识符的命名规范，且最大长度不超过63个字符。

- **password**

  登录密码。

  密码规则如下：

  - 密码默认不少于8个字符。
  - 不能与用户名及用户名倒序相同。
  - 至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。使用范围外的字符会收到告警，但依然允许创建。

  取值范围：字符串。

- **DISABLE**

  默认情况下，用户可以更改自己的密码，除非密码被禁用。要禁用用户的密码，请指定DISABLE。禁用某个用户的密码后，将从系统中删除该密码，此类用户只能通过外部认证来连接数据库，例如：IAM认证、kerberos或ldap认证。只有管理员才能启用或禁用密码。普通用户不能禁用初始用户的密码。要启用密码，请运行ALTER USER并指定密码。

- **ENCRYPTED | UNENCRYPTED**

  控制密码存储在系统表里的密码是否加密。（如果没有指定，那么缺省的行为由配置参数password_encryption_type控制。）按照产品安全要求，密码必须加密存储，所以，UNENCRYPTED在GaussDB(DWS)中禁止使用。因为系统无法对指定的加密密码字符串进行解密，所以如果目前的密码字符串已经是用SHA256加密的格式，则会继续照此存放，而不管是否声明了ENCRYPTED或UNENCRYPTED。这样就允许在dump/restore的时候重新加载加密的密码。

- **SYSADMIN | NOSYSADMIN**

  决定一个新用户是否为“系统管理员”，具有SYSADMIN属性的用户拥有系统最高权限。

  缺省为NOSYSADMIN。

- **AUDITADMIN | NOAUDITADMIN**

  定义用户是否有审计管理属性。

  缺省为NOAUDITADMIN。

- **CREATEDB | NOCREATEDB**

  决定一个新用户是否能创建数据库。

  新用户没有创建数据库的权限。缺省为NOCREATEDB。

- **USEFT | NOUSEFT**

  决定一个新用户是否能操作外表，包括：新建外表、删除外表、修改外表、读写外表。

  新用户没有操作外表的权限。

  缺省为NOUSEFT。

- **CREATEROLE | NOCREATEROLE**

  决定一个用户是否可以创建新角色或用户（也就是执行CREATE ROLE和CREATE USER）。 一个拥有CREATEROLE权限的用户也可以修改和删除其他用户或角色。

缺省为NOCREATEROLE。

- **INHERIT | NOINHERIT**

  这些子句决定一个用户是否“继承”它所在组的用户的权限。不推荐使用。

- **LOGIN | NOLOGIN**

  具有LOGIN属性的用户才可以登录数据库。

  缺省为LOGIN。

- **REPLICATION | NOREPLICATION**

  定义用户是否允许流复制或设置系统为备份模式。REPLICATION属性是特定的用户，仅用于复制。

  缺省为NOREPLICATION。

- **INDEPENDENT | NOINDEPENDENT**

  定义私有、独立的用户。具有INDEPENDENT属性的用户，管理员对其进行的控制、访问的权限被分离，具体规则如下：

  - 未经INDEPENDENT用户授权，管理员无权对其表对象进行增、删、查、改、复制、授权操作。
  - 未经INDEPENDENT用户授权，管理员无权修改INDEPENDENT用户的继承关系。
  - 管理员无权修改INDEPENDENT用户的表对象的属主。
  - 管理员无权去除INDEPENDENT用户的INDEPENDENT属性。
  - 管理员无权修改INDEPENDENT用户的数据库密码，INDEPENDENT用户需管理好自身密码，密码丢失无法重置。
  - 管理员属性用户不允许定义修改为INDEPENDENT属性。

- **VCADMIN | NOVCADMIN**

  定义逻辑集群管理员用户。具有逻辑集群管理员属性的用户，和普通用户相比，有如下额外权限：

  - 在所关联逻辑集群中创建、修改和删除资源池的权限。
  - 将所关联的逻辑集群的访问权限授予其他用户或角色，或回收其他用户或角色对关联逻辑集群的访问权限。

- **CONNECTION LIMIT**

  声明该用户在单个CN上可以使用的并发连接数量。

  取值范围：整数，>=-1，缺省值为-1，表示没有限制。

  须知

  为保证集群正常使用，connection limit的最小值是集群中CN的数目。在集群做ANALYZE时，其他CN节点会连接当前做ANALYZE的CN节点来同步元数据。例如集群中有3个CN节点，那么connection limit应该设置为>=3。

- **VALID BEGIN**

  设置用户生效的时间戳。如果省略了该子句，用户无有效开始时间限制。

- **VALID UNTIL**

  设置用户失效的时间戳。如果省略了该子句，用户无有效结束时间限制。

- **RESOURCE POOL**

设置用户使用的resource pool名字，该名字属于系统表：pg_resource_pool。

- **USER GROUP 'groupuser'**

  创建一个user的子用户。

- **PERM SPACE**

  设置用户永久表存储空间限额。

  space_limit：永久表存储空间上限。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **TEMP SPACE**

  设置用户临时表存储空间限额。

  tmpspacelimit：临时表存储空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **SPILL SPACE**

  设置用户算子落盘空间限额。

  spillspacelimit：算子落盘空间限额。取值范围：字符串格式为正整数+单位，单位当前支持K/M/G/T/P。0表示不限制。

- **NODE GROUP**

  设置用户关联的逻辑集群名称。如果需要关联的逻辑集群名称包含大写字符或特殊字符，指定逻辑集群名称时需要加双引号。

- **IN ROLE**

  新用户立即拥有IN ROLE子句中列出的一个或多个现有用户拥有的权限。不推荐使用。

- **IN GROUP**

  IN GROUP是IN ROLE过时的拼法。不推荐使用。

- **ROLE**

  ROLE子句列出一个或多个现有的用户，它们将自动添加为这个新用户的成员，拥有新用户所有的权限。

- **ADMIN**

  ADMIN子句类似ROLE子句，不同的是ADMIN后的用户可以把新用户的权限赋给其他用户。

- **USER**

  USER子句是ROLE子句过时的拼法。

- **SYSID**

  SYSID子句将被忽略，无实际意义。

- **DEFAULT TABLESPACE**

  DEFAULT TABLESPACE子句将被忽略，无实际意义。

- **PROFILE**

  PROFILE子句将被忽略，无实际意义。

- **PGUSER**

  该属性用于兼容开源Postgres的连接通讯，开源的Postgres客户端接口（推荐使用Postgres 9.2.19版本的相关客户端接口）可以使用具有该属性的数据库用户连接数据库。

须知

该属性只用于兼容连接过程，而由于本产品与Postgres的内核差异导致的不兼容，不在此属性控制范围内。

由于具有PGUSER属性的用户的认证方式与其他用户不同，开源客户端的报错信息可能导致数据库用户PGUSER属性被枚举，建议使用本产品自有的客户端。例如：

```
#normaluser是不具有PGUSER属性的用户，psql是Postgres的客户端工具
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U normaluser
psql: authentication method 10 not supported

#pguser用户是具有PGUSER属性的用户
pg@dws04:~> psql -d postgres -p 8000 -h 10.11.12.13 -U pguser
Password for user pguser:
```

- **AUTHINFO 'authinfo'**

  该属性用于指定用户认证类型，authinfo为类型说明字符串，大小写敏感。当前仅支持LDAP类型，对应的类型说明字符串为'ldap'。LDAP属于外部认证，故需要同时指定PASSWORD DISABLE。

  须知

  - authinfo可以加上LDAP认证的额外信息，比如LDAP认证中的fulluser，fulluser等同于ldapprefix+username+ldapsuffix。当authinfo为'ldap'，表示用户认证类型为LDAP，此时ldapprefix和ldapsuffix的信息由pg_hba.conf中匹配的记录提供。
  - ALTER ROLE时不允许用户切换认证类型，仅允许LDAP用户修改LDAP属性。

- **PASSWORD EXPIRATION period**

  声明该用户的登录密码过期天数，登录密码过期之前用户需要及时修改密码。登录密码过期后用户无法登录，需要请管理员设置新的登录密码后登录。

  取值范围：整数，-1~999。缺省值为-1，表示没有过期限制；0表示用户登录密码立即过期。

示例

创建用户jim：

```
CREATE USER jim PASSWORD '{Password}';
```

下面语句与上面的等价：

```
CREATE USER kim IDENTIFIED BY '{Password}';
```

如果创建有“创建数据库”权限的用户，则需要加CREATEDB关键字：

```
CREATE USER dim CREATEDB PASSWORD '{Password}';
```

相关链接

ALTER USER，**DROP USER**

12.58 CREATE VIEW

功能描述

创建一个视图。视图与基本表不同，是一个虚拟的表。数据库中仅存放视图的定义，而不存放视图对应的数据，这些数据仍存放在原来的基本表中。若基本表中的数据发生变化，从视图中查询出的数据也随之改变。从这个意义上讲，视图就像一个窗口，透过它可以看到数据库中用户感兴趣的数据及变化。

注意事项

- 视图依赖的基表重命名之后，需要将视图手动重建。
- 创建视图时使用WITH(security_barriers)可以创建一个相对安全的视图，避免攻击者利用低成本函数的RAISE语句打印出隐藏的基表数据。
- GUC参数view_independent打开时，支持普通视图删除列。需注意，如果存在列级约束，不支持该列的删除。

语法格式

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW view_name [ ( column_name [, ...] ) ]
    [ WITH ( {view_option_name [= view_option_value]} [, ... ] ) ]
    AS query;
```

参数说明

- **OR REPLACE**

  如果视图已存在，则重新定义。

- **TEMP | TEMPORARY**

  创建临时视图。

- **view_name**

  要创建的视图名称。可以用模式修饰。

  取值范围：字符串，符合标识符命名规范。

- **column_name**

  可选的名字列表，用作视图的字段名。如果没有给出，字段名取自查询中的字段名。

  取值范围：字符串，符合标识符命名规范。

- **view_option_name [= view_option_value]**

  该子句为视图指定一个可选的参数。

  目前view_option_name支持的参数仅有security_barrier，当VIEW试图提供行级安全时，应使用该参数。

  取值范围：boolean类型，TRUE、FALSE

- **query**

  为视图提供行和列的SELECT或VALUES语句。

须知

视图解耦功能下不支持CTE重名。例如：

```
CREATE TABLE t1(a1 INT, b1 INT);
CREATE TABLE t2(a2 INT, b2 INT, c2 INT);
CREATE OR REPLACE VIEW v1 AS WITH tmp AS (SELECT * FROM t2) ,tmp1 AS (SELECT b2,c2 FROM tmp WHERE b2 = (WITH RECURSIVE tmp(aa, bb) AS (SELECT a1,b1 FROM t1) SELECT bb FROM tmp WHERE aa = c2)) SELECT c2 FROM tmp1;
```

示例

创建字段spcname为pg_default组成的视图：

```
CREATE VIEW myView AS
    SELECT * FROM pg_tablespace WHERE spcname = 'pg_default';
```

对已存在视图myView进行重新定义，创建字段spcname为pg_global组成的视图：

```
CREATE OR REPLACE VIEW myView AS
    SELECT * FROM pg_tablespace WHERE spcname = 'pg_global';
```

创建一个由c_customer_sk小于150的内容组成的视图：

```
CREATE VIEW tpcds.customer_details_view_v1 AS
    SELECT * FROM tpcds.customer
    WHERE c_customer_sk < 150;
```

可更新的视图

当开启视图可更新参数（enable_view_update）后，系统允许对简单视图使用INSERT，UPDATE、DELETE和MERGE INTO语句进行更新。（MERGE INTO语句更新仅8.1.2及以上版本支持）

满足以下所有条件的视图可进行更新：

- 视图定义的FROM语句中只能有一个普通表，不能是系统表、外表、delta表、toast表、错误表。
- 视图中包含可更新的列，这些列是对基础表可更新列的简单引用。
- 视图定义不能包含WITH、DISTINCT、GROUP BY、ORDER BY、FOR UPDATE、FOR SHARE、HAVING、TABLESAMPLE、LIMIT、OFFSET子句。
- 视图定义不能包含UNION、INTERSECT、EXCEPT集合操作。
- 视图定义的选择列表不能包含聚集函数、窗口函数、返回集合的函数。
- 对于INSERT、UPDATE和DELETE语句，视图上不能有触发时机为INSTEAD OF的触发器。对于MERGE INTO语句，视图和基础表上都不能有触发器。
- 视图定义不能包含子链接。
- 视图定义不能包含属性为VOLATILE的函数（函数值可以在一次表扫描内改变的函数）
- 视图定义不能对表的分布键所在列起别名，或将普通列起别名为分布键列名。
- 视图更新操作中包含RETURNING子句时，视图定义中的列只能来自于基础表。

如果可更新的视图定义包含WHERE条件，则该条件将会限制UPDATE和DELETE语句修改基础表上的行。如果UPDATE语句更改行后不再满足WHERE条件，更新后通过视图将无法查询到。类似地如果INSERT命令插入了不满足WHERE条件的数据，插入后通过视图将无法查询到。在视图上执行插入、更新或删除的用户必须在视图和表上具有相应的插入、更新或删除权限。

相关链接

ALTER VIEW，**DROP VIEW**

12.59 CURSOR

功能描述

CURSOR命令定义一个游标，用于在一个大的查询里面检索少数几行数据。

为了处理SQL语句，存储过程进程分配一段内存区域来保存上下文联系。游标是指向上下文区域的句柄或指针。借助游标，存储过程可以控制上下文区域的变化。

注意事项

- 游标命令只能在事务块里使用。
- 通常游标和SELECT一样返回文本格式。因为数据在系统内部是用二进制格式存储的，系统必须对数据做一定转换以生成文本格式。一旦数据是以文本形式返回，客户端应用需要把它们转换成二进制进行操作。使用FETCH语句，游标可以返回文本或二进制格式。
- 需小心使用二进制游标。文本格式一般都比对应的二进制格式占用的存储空间大。二进制游标返回内部二进制形态的数据，可能更易于操作。如果想以文本方式显示数据，则以文本方式检索会为用户节约很多客户端的工作。比如，如果查询从某个整数列返回1，在缺省的游标里将获得一个字符串1，但在二进制游标里将得到一个4字节的包含该数值内部形式的数值（大端顺序）。

语法格式

```
CURSOR cursor_name
    [ BINARY ]  [ NO SCROLL ]  [ { WITH | WITHOUT } HOLD ]
    FOR query ;
```

参数说明

- **cursor_name**

  将要创建的游标名。

  取值范围：遵循数据库对象命名规范。

- **BINARY**

  指明游标以二进制而不是文本格式返回数据。

- **NO SCROLL**

  声明游标检索数据行的方式。

  - NO SCROLL：声明该游标不能用于以倒序的方式检索数据行。
  - 未声明：根据执行计划的不同，自动判断该游标是否可以用于以倒序的方式检索数据行。

- **WITH HOLD | WITHOUT HOLD**

  声明当创建游标的事务结束后，游标是否能继续使用。

  - WITH HOLD：声明该游标在创建它的事务结束后仍可继续使用。
  - WITHOUT HOLD：声明该游标在创建它的事务之外不能再继续使用，此游标将在事务结束时被自动关闭。

  - 如果不指定WITH HOLD或WITHOUT HOLD，默认行为是WITHOUT HOLD。

- **query**

  使用SELECT或VALUES子句指定游标返回的行。

  取值范围：SELECT或VALUES子句。

示例

开启事务：

```
START TRANSACTION;
```

创建名称为cursor1的游标：

```
CURSOR cursor1 FOR SELECT * FROM tpcds.customer_address ORDER BY 1;
```

创建名称为cursor2的游标：

```
CURSOR cursor2 FOR VALUES(1,2),(0,3) ORDER BY 1;
```

WITH HOLD游标的使用示例：

1. 创建一个with hold游标：
   ```
   DECLARE cursor3 CURSOR WITH HOLD FOR SELECT * FROM tpcds.customer_address ORDER BY 1;
   ```
2. 抓取头2行到游标cursor3里：
   ```
   FETCH FORWARD 2 FROM cursor3;
   ```
3. 结束事务：
   ```
   END;
   ```
4. 抓取下一行到游标cursor3里：
   ```
   FETCH FORWARD 1 FROM cursor3;
   ```
5. 关闭游标：
   ```
   CLOSE cursor3;
   ```

相关链接

FETCH

12.60 DROP DATABASE

功能描述

删除一个数据库。

注意事项

- 只有数据库所有者有权限执行DROP DATABASE命令，系统管理员默认拥有此权限。
- 不能对系统默认安装的三个数据库（gaussdb、TEMPLATE0和TEMPLATE1）执行删除操作，系统做了保护。如果想查看当前服务中有哪几个数据库，可以用gsql的\l命令查看。
- 如果有用户正在与要删除的数据库连接，则删除操作失败。如果要查看当前存在哪些数据库连接，可以通过视图v$session查看。
- 不能在事务块中执行DROP DATABASE命令。

- 如果执行DROP DATABASE失败，事务回滚，需要再次执行一次DROP DATABASE IF EXISTS。
- DROP DATABASE一旦执行将无法撤销，请谨慎使用。
- DROP DATABASE若提示database is being accessed by other users类错误，可能原因为CLEAN CONNECTION过程存在线程无法及时响应信号，出现连接清理不完全的情况，需要再次执行CLEAN CONNECTION。

语法格式

```
DROP DATABASE [ IF EXISTS ] database_name ;
```

参数说明

- **IF EXISTS**

  如果指定的数据库不存在，则发出一个notice而不是抛出一个错误。
- **database_name**

  要删除的数据库名称。

  取值范围：字符串，已存在的数据库名称。

示例

删除名称为music的数据库：

```
DROP DATABASE music;
```

相关链接

CREATE DATABASE，**ALTER DATABASE**

12.61 DROP FOREIGN TABLE

功能描述

删除指定的外表。

注意事项

DROP FOREIGN TABLE会强制删除指定的外表，删除外表后，依赖该外表的索引会被删除，而使用到该外表的函数和存储过程将无法执行。

语法格式

```
DROP FOREIGN TABLE [ IF EXISTS ]
    table_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**

  如果指定的外表不存在，则发出一个notice而不是抛出一个错误。
- **table_name**

  要删除的外表名称。

取值范围：已存在的外表名。

- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖于外表的对象（比如视图）。
  - RESTRICT：如果存在依赖对象，则拒绝删除该外表（缺省行为）。。

示例

删除名称为customer_ft的外表：

```
DROP FOREIGN TABLE customer_ft;
```

相关链接

ALTER FOREIGN TABLE (GDS导入导出)，**ALTER FOREIGN TABLE (For HDFS or OBS)**，**CREATE FOREIGN TABLE (GDS导入导出)**，**CREATE FOREIGN TABLE (SQL on OBS or Hadoop)**

12.62 DROP FUNCTION

功能描述

删除一个已存在的函数。

注意事项

- 如果所删除的函数为重载函数，则删除时需要指明该函数的参数类型，如为非重载函数，则可直接指定函数名进行删除。
- 如果函数中涉及对临时表相关操作，则无法使用DROP FUNCTION删除函数。

语法格式

```
DROP FUNCTION [ IF EXISTS ] function_name
[ ( [ {[ argmode ] [ argname ] argtype} [, ...] ] ) [ CASCADE | RESTRICT ] ];
```

参数说明

- **IF EXISTS**

  如果指定的函数不存在，则发出一个notice而不是抛出一个错误。
- **function_name**

  要删除的函数名。

  取值范围：已存在的函数名。
- **argmode**

  函数参数的模式。
- **argname**

  函数参数的名称。
- **argtype**

  函数参数的数据类型。
- **CASCADE | RESTRICT**

- CASCADE：级联删除依赖于函数的对象（比如操作符）。
- RESTRICT：如果有任何依赖对象存在，则拒绝删除该函数（缺省行为）。

示例

删除名称为add_two_number的函数：

```
DROP FUNCTION add_two_number;
```

相关链接

ALTER FUNCTION，**CREATE FUNCTION**

12.63 DROP GROUP

功能描述

删除用户组。

DROP GROUP是DROP ROLE的别名。

注意事项

DROP GROUP是集群管理工具封装的内部接口，用来实现集群管理。该接口不建议用户直接使用，以免对集群状态造成影响。

语法格式

```
DROP GROUP [ IF EXISTS ] group_name [, ...];
```

参数说明

请参见DROP ROLE的**参数说明**。

相关链接

CREATE GROUP，**ALTER GROUP**，**DROP ROLE**

12.64 DROP INDEX

功能描述

删除索引。

注意事项

只有索引的所有者有权限执行DROP INDEX命令，系统管理员默认拥有此权限。

语法格式

```
DROP INDEX  [ CONCURRENTLY ] [ IF EXISTS ]
  index_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **CONCURRENTLY**

  删除索引而不锁定索引表上的并发选择，插入，更新和删除。 普通的DROP INDEX在表上获取排他锁，从而阻止其他访问，直到可以完成索引删除为止。 使用此选项，命令将一直等到冲突的事务完成。

  使用此选项时需要注意：只能指定一个索引名称，并且不支持CASCADE选项。（ 因此，不能以这种方式删除支持UNIQUE或PRIMARY KEY约束的索引。 ）可以在事务块内执行常规的DROP INDEX命令，但不能以DROP INDEX CONCURRENTLY方式执行。

- **IF EXISTS**

  如果指定的索引不存在，则发出一个notice而不是抛出一个错误。

- **index_name**

  要删除的索引名。

  取值范围：已存在的索引。

- **CASCADE | RESTRICT**
  - CASCADE：表示允许级联删除依赖于该索引的对象。
  - RESTRICT：缺省值，表示有依赖与此索引的对象存在，则该索引无法被删除。

示例

删除现有的索引ds_ship_mode_t1_index2：

```
DROP INDEX tpcds.ds_ship_mode_t1_index2;
```

相关链接

ALTER INDEX，**CREATE INDEX**

12.65 DROP OWNED

功能描述

删除一个数据库角色所拥有的数据库对象。

注意事项

所有该角色在当前数据库里和共享对象（ 数据库，表空间 ）上的所有对象上的权限都将被撤销。

语法格式

```
DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **name**

  将要删除所拥有对象并且撤销其权限的角色名。

- **CASCADE | RESTRICT**
  - CASCADE：级联删除所有依赖于被删除对象的对象。
  - RESTRICT：缺省值，拒绝删除那些有任何依赖对象存在的对象。

示例

删除名称为u1的角色拥有的所有数据库对象：

```
DROP OWNED BY u1;
```

12.66 DROP REDACTION POLICY

功能描述

删除应用在指定表上的脱敏策略。

注意事项

只有表的属主才有权限删除脱敏策略。

语法格式

```
DROP REDACTION POLICY [ IF EXISTS ] policy_name ON table_name;
```

参数说明

- **IF EXISTS**
  如果待删除的脱敏策略不存在，则发出一个NOTICE，而不是抛出一个错误。
- **policy_name**
  要删除的脱敏策略名称。
- **table_name**
  要删除的脱敏策略所应用的表名。

示例

删除表emp上的脱敏策略mask_emp：

```
DROP REDACTION POLICY mask_emp ON emp;
```

相关链接

ALTER REDACTION POLICY，**CREATE REDACTION POLICY**

12.67 DROP ROW LEVEL SECURITY POLICY

功能描述

删除表上指定的行访问控制策略。

注意事项

仅表的所有者或者管理员用户才能删除表的行访问控制策略。

语法格式

```
DROP [ ROW LEVEL SECURITY ] POLICY [ IF EXISTS ] policy_name ON table_name [ CASCADE | RESTRICT ]
```

参数说明

- **IF EXISTS**
  如果指定的行访问控制策略不存在，发出一个notice而不是抛出一个错误。
- **policy_name**
  要删除的行访问控制策略的名称。
  - table_name
    行访问控制策略所在的数据表名。
  - CASCADE/RESTRICT
    仅适配此语法，无对象依赖于该行访问控制策略，CASCADE和RESTRICT效果相同。

示例

删除表all_data上的行访问控制策略all_data_rls：

```
DROP ROW LEVEL SECURITY POLICY all_data_rls ON all_data;
```

相关链接

ALTER ROW LEVEL SECURITY POLICY，**CREATE ROW LEVEL SECURITY POLICY**

12.68 DROP PROCEDURE

功能描述

删除已存在的存储过程。

注意事项

无。

语法格式

```
DROP PROCEDURE [ IF EXISTS ] procedure_name ;
```

参数说明

- **IF EXISTS**
  如果指定的存储过程不存在，发出一个notice而不是抛出一个错误。
- **procedure_name**
  要删除的存储过程名字。

取值范围：已存在的存储过程名。

示例

删除存储过程：

```
DROP PROCEDURE prc_add;
```

相关链接

CREATE PROCEDURE

12.69 DROP RESOURCE POOL

功能描述

删除一个资源池。

说明

如果某个角色已关联到该资源池，则无法删除该资源池。

注意事项

只要用户对当前数据库有DROP权限，就可以删除资源池。

语法格式

```
DROP RESOURCE POOL [ IF EXISTS ] pool_name;
```

参数说明

- **IF EXISTS**
  如果指定的存储过程不存在，发出一个notice而不是抛出一个错误。
- **pool_name**
  已创建过的资源池名称。
  取值范围：字符串，要符合标识符的命名规范。

示例

删除资源池pool：

```
DROP RESOURCE POOL pool;
```

相关链接

ALTER RESOURCE POOL，**CREATE RESOURCE POOL**

12.70 DROP ROLE

功能描述

删除指定的角色。

注意事项

DROP ROLE若提示role is being used by other users错误，可能原因为CLEAN CONNECTION过程存在线程无法及时响应信号，出现连接清理不完全的情况，需要再次执行CLEAN CONNECTION。

语法格式

```
DROP ROLE [ IF EXISTS ] role_name [, ...];
```

参数说明

- **IF EXISTS**

  如果指定的角色不存在，则发出一个notice而不是抛出一个错误。

- **role_name**

  要删除的角色名称。

  取值范围：已存在的角色。

示例

删除角色manager：

```
DROP ROLE manager;
```

相关链接

CREATE ROLE，**ALTER ROLE**，**SET ROLE**

12.71 DROP SCHEMA

功能描述

从数据库中删除模式。

注意事项

- 只有模式的所有者或者被授予了模式DROP权限的用户有权限执行DROP SCHEMA命令，系统管理员默认拥有此权限。
- 不得随意删除pg_temp或pg_toast_temp开头的模式，这些模式是系统内部使用的，如果删除，可能导致无法预知的结果。
- 无法删除当前模式。如果要删除当前模式，须切换到其他模式下。

语法格式

```
DROP SCHEMA [ IF EXISTS ] schema_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**
  如果指定的模式不存在，发出一个notice而不是抛出一个错误。
- **schema_name**
  模式的名字。
  取值范围：已存在模式名。
- **CASCADE | RESTRICT**
  - CASCADE：自动删除包含在模式中的对象。
  - RESTRICT：如果模式包含任何对象，则删除失败（缺省行为）。

示例

删除模式ds_new：

```
DROP SCHEMA ds_new;
```

相关链接

ALTER SCHEMA，**CREATE SCHEMA**。

12.72 DROP SEQUENCE

功能描述

从当前数据库里删除序列。

注意事项

只有序列的所有者或者系统管理员才能删除。

语法格式

```
DROP SEQUENCE [ IF EXISTS ] {[schema.]sequence_name} [ , ... ] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**
  如果指定的序列不存在，则发出一个notice而不是抛出一个错误。
- **name**
  序列名称。
- **CASCADE**
  级联删除依赖序列的对象。
- **RESTRICT**
  如果存在任何依赖的对象，则拒绝删除序列。此项是缺省值。

示例

删除序列serial：

```
DROP SEQUENCE serial;
```

相关链接

CREATE SEQUENCE ALTER SEQUENCE

12.73 DROP SERVER

功能描述

删除现有的一个数据服务器。

注意事项

只有server的owner才可以删除。

语法格式

```
DROP SERVER [ IF EXISTS ] server_name [ {CASCADE | RESTRICT} ] ;
```

参数描述

- **IF EXISTS**
  如果指定的表不存在，则发出一个notice而不是抛出一个错误。
- **server_name**
  服务器名称。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖于server的对象 。
  - RESTRICT（缺省值）：如果存在依赖对象，则拒绝删除该server。

示例

删除hdfs_server：

```
DROP SERVER hdfs_server;
```

相关链接

CREATE SERVER，**ALTER SERVER**

12.74 DROP SYNONYM

功能描述

删除指定的SYNONYM对象。

注意事项

只有SYNONYM的所有者有权限执行DROP SYNONYM命令，系统管理员默认拥有此权限。

语法格式

```
DROP SYNONYM [ IF EXISTS ] synonym_name [ CASCADE | RESTRICT ];
```

参数描述

- **IF EXISTS**
  如果指定的同义词不存在，则发出一个notice而不是抛出一个错误。
- **synonym_name**
  同义词名字，可以带模式名。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖同义词的对象（比如视图）。
  - RESTRICT：如果有依赖对象存在，则拒绝删除同义词。此选项为缺省值。

示例

删除同义词：

```
DROP SYNONYM t1;
```

相关链接

ALTER SYNONYM，**CREATE SYNONYM**

12.75 DROP TABLE

功能描述

删除指定的表。

注意事项

- 只有表的所有者、模式所有者或者被授予了表的DROP权限的用户才能执行DROP TABLE，系统管理员默认拥有该权限。要清空指定表中的行但是不删除该表定义，可以使用TRUNCATE或者DELETE。
- DROP TABLE会强制删除指定的表，删除表后，依赖该表的索引会被删除，而使用到该表的函数和存储过程将无法执行。删除分区表，会同时删除分区表中的所有分区。

语法格式

```
DROP TABLE [ IF EXISTS ]
  { [schema.]table_name } [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**

如果指定的表不存在，则发出一个notice而不是抛出一个错误。

- **schema**
  模式名称。
- **table_name**
  表名称。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖于表的对象（比如视图）。
  - RESTRICT（缺省项）：如果存在依赖对象，则拒绝删除该表。这个是缺省。

示例

删除表warehouse_t1：

```
DROP TABLE tpcds.warehouse_t1;
```

相关链接

ALTER TABLE，**12.101-RENAME TABLE**，**CREATE TABLE**

12.76 DROP TEXT SEARCH CONFIGURATION

功能描述

删除已有文本搜索配置。

注意事项

要执行DROP TEXT SEARCH CONFIGURATION命令，用户必须是该文本搜索配置的所有者。

语法格式

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**
  如果指定的文本搜索配置不存在，那么发出一个notice而不是抛出一个错误。
- **name**
  要删除的文本搜索配置名称（可有模式修饰）。
- **CASCADE**
  级联删除依赖文本搜索配置的对象。
- **RESTRICT**
  若有任何对象依赖文本搜索配置则拒绝删除它。这是默认情况。

示例

删除文本搜索配置ngram1:

```
DROP TEXT SEARCH CONFIGURATION ngram1;
```

相关链接

ALTER TEXT SEARCH CONFIGURATION， CREATE TEXT SEARCH CONFIGURATION

12.77 DROP TEXT SEARCH DICTIONARY

功能描述

删除全文检索词典。

注意事项

- 预定义词典不支持DROP操作。
- 只有词典的所有者可以执行DROP操作，系统管理员默认拥有此权限。
- 谨慎执行DROP...CASCADE操作，该操作将级联删除使用该词典的文本搜索配置（TEXT SEARCH CONFIGURATION）。

语法格式

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

参数说明

- **IF EXISTS**

  如果指定的全文检索词典不存在，那么发出一个Notice而不是报错。
- **name**

  要删除的词典名称（可指定模式名，否则默认在当前模式下）。

  取值范围：已存在的词典名。
- **CASCADE**

  自动删除依赖于该词典的对象，并依次删除依赖于这些对象的所有对象。

  如果存在任何一个使用该词典的文本搜索配置，此DROP命令将不会成功。可添加CASCADE以删除引用该词典的所有文本搜索配置以及词典。
- **RESTRICT**

  如果任何对象依赖词典，则拒绝删除该词典。这是缺省值。

示例

删除词典english：

```
DROP TEXT SEARCH DICTIONARY english;
```

相关链接

ALTER TEXT SEARCH DICTIONARY，CREATE TEXT SEARCH DICTIONARY

12.78 DROP TRIGGER

功能描述

删除触发器。

注意事项

只有触发器的所有者可以执行DROP TRIGGER操作，系统管理员默认拥有此权限。

语法格式

```
DROP TRIGGER [ IF EXISTS ] trigger_name ON table_name [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**
  如果指定的触发器不存在，则发出一个notice而不是抛出一个错误。
- **trigger_name**
  要删除的触发器名字。
  取值范围：已存在的触发器。
- **table_name**
  要删除的触发器所在的表名称。
  取值范围：已存在的含触发器的表。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖此触发器的对象。
  - RESTRICT：如果有依赖对象存在，则拒绝删除此触发器。此选项为缺省值。

示例

删除触发器insert_trigger：

```
DROP TRIGGER insert_trigger ON test_trigger_src_tbl;
```

相关链接

CREATE TRIGGER，**ALTER TRIGGER**

12.79 DROP TYPE

功能描述

删除一个用户定义的数据类型。只有类型所有者才有删除权限。

语法格式

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

参数说明

- **IF EXISTS**

  如果指定的类型不存在，那么发出一个notice而不是抛出一个错误。
- **name**

  要删除的类型名(可以有模式修饰)。
- **CASCADE**

  级联删除依赖该类型的对象(比如字段、函数、操作符等)

  RESTRICT

  如果有依赖对象，则拒绝删除该类型（缺省行为）。

示例

删除compfoo类型：

```
DROP TYPE compfoo cascade;
```

相关链接

ALTER TYPE，**CREATE TYPE**

12.80 DROP USER

功能描述

删除用户，同时会删除同名的schema。

注意事项

- 须使用CASCADE级联删除依赖用户的对象（除数据库外）。当删除用户的级联对象时，如果级联对象处于锁定状态，则此级联对象无法被删除，直到对象被解锁或锁定级联对象的进程被终止。
- 在数据库中删除用户时，如果依赖用户的对象在其他数据库中或者依赖用户的对象是其他数据库，请用户先手动删除其他数据库中的依赖对象或直接删除依赖数据库，再删除用户。即DROP USER不支持跨数据库进行级联删除。
- 在多租户场景下，删除组用户时，业务用户也会同时被删除，如果指定CASCADE级联删除，那么删除业务用户时同时也指定CASCADE。如果在删除某个用户失败时，同时其他用户也无法成功删除。
- 如果用户A创建的GDS外表指定的错误表在用户B的schema下，则无法通过DROP USER指定CASCADE关键字直接删除用户B。
- DROP USER若提示role is being used by other users错误，可能原因为CLEAN CONNECTION过程存在线程无法及时响应信号，出现连接清理不完全的情况，需要再次执行CLEAN CONNECTION。

语法格式

```
DROP USER [ IF EXISTS ] user_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**
  如果指定的用户不存在，发出一个notice而不是抛出一个错误。
- **user_name**
  待删除的用户名。
  取值范围：已存在的用户名。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖用户的表等对象。级联删除用户的时候，会删除掉owner是这个用户的对象，并清理掉其他对象对这个用户的授权信息。
  - RESTRICT：如果用户还有任何依赖的对象，则拒绝删除该用户（缺省行为）。

  说明

  在GaussDB(DWS)中，存在一个配置参数enable_kill_query，此参数在配置文件postgresql.conf中。此参数影响级联删除用户对象的行为：
  - 当参数enable_kill_query为on ，且使用CASCADE模式删除用户时，会自动kill锁定用户级联对象的进程，并删除用户。
  - 当参数enable_kill_query为off，且使用CASCADE模式删除用户时，会等待锁定级联对象的进程结束之后再删除用户。

示例

删除用户jim：

```
DROP USER jim CASCADE;
```

相关链接

ALTER USER，**CREATE USER**

12.81 DROP VIEW

功能描述

数据库中强制删除已有的视图。

注意事项

- 只有视图的所有者有权限执行DROP VIEW的命令，系统管理员默认拥有此权限。
- 数据库中仅存放视图的定义，而不存放视图对应的数据，在基表未变的的前提下，误操作删除的视图可通过**CREATE VIEW**来重建视图。

语法格式

```
DROP VIEW [ IF EXISTS ] view_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**
  如果指定的视图不存在，则发出一个notice而不是抛出一个错误。

- **view_name**
  要删除的视图名字。
  取值范围：已存在的视图。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖此视图的对象（比如其他视图）。
  - RESTRICT：如果有依赖对象存在，则拒绝删除此视图。此选项为缺省值。

示例

删除视图myView：

```
DROP VIEW myView;
```

删除视图customer_details_view_v2：

```
DROP VIEW public.customer_details_view_v2;
```

相关链接

ALTER VIEW，**CREATE VIEW**

12.82 FETCH

功能描述

FETCH通过已创建的游标来检索数据。

每个游标都有一个供FETCH使用的关联位置。游标的关联位置可以在查询结果的第一行之前，或者在结果中的任意行，或者在结果的最后一行之后：

- 游标刚创建完之后，关联位置在第一行之前。
- 在抓取了一些移动行之后，关联位置在检索到的最后一行上。
- 如果FETCH抓取完了所有可用行，它就停在最后一行后面，或者在反向抓取的情况下是停在第一行前面。
- FETCH ALL或FETCH BACKWARD ALL将总是把游标的关联位置放在最后一行或者在第一行前面。

注意事项

- 如果游标定义了NO SCROLL，则不允许使用例如FETCH BACKWARD之类的反向抓取。
- NEXT，PRIOR，FIRST，LAST，ABSOLUTE，RELATIVE形式在恰当地移动游标之后抓取一条记录。如果后面没有数据行，就返回一个空的结果，此时游标就会停在查询结果的最后一行之后（向后查询时）或者第一行之前（向前查询时）。
- FORWARD和BACKWARD形式在向前或者向后移动的过程中抓取指定的行数，然后把游标定位在最后返回的行上；或者是，如果count大于可用的行数，则在所有行之后（向后查询时）或者之前（向前查询时）。
- RELATIVE 0，FORWARD 0，BACKWARD 0都要求在不移动游标的前提下抓取当前行，也就是重新抓取最近刚抓取过的行。除非游标定位在第一行之前或者最后一行之后，这个动作都应该成功，而在那两种情况下，不返回任何行。

- 当FETCH的游标上涉及列存表时，不支持BACKWARD、PRIOR、FIRST等涉及反向获取操作。

语法格式

```
FETCH [ direction { FROM | IN } ] cursor_name;
```

其中direction子句为可选参数。

```
NEXT
  | PRIOR
  | FIRST
  | LAST
  | ABSOLUTE count
  | RELATIVE count
  | count
  | ALL
  | FORWARD
  | FORWARD count
  | FORWARD ALL
  | BACKWARD
  | BACKWARD count
  | BACKWARD ALL
```

参数说明

- **direction_clause**

  定义抓取数据的方向。

  取值范围：

  - NEXT（缺省值）

    从当前关联位置开始，抓取下一行。

  - PRIOR

    从当前关联位置开始，抓取上一行。

  - FIRST

    抓取查询的第一行（和ABSOLUTE 1相同）。

  - LAST

    抓取查询的最后一行（和ABSOLUTE -1相同）。

  - ABSOLUTE count

    抓取查询中第count行。

    ABSOLUTE抓取不会比用相对位移移动到需要的数据行更快，因为下层的实现必须遍历所有中间的行。

    count取值范围：有符号的整数

    - count为正数，就从查询结果的第一行开始，抓取第count行。当count小于当前游标位置时，涉及到rewind操作，暂不支持。
    - count为负数或0，涉及到反向扫描操作，暂不支持。

  - RELATIVE count

    从当前关联位置开始，抓取随后或前面的第count行。

    取值范围：有符号的整数

    - count为正数就抓取当前关联位置之后的第count行。

    - count为负数，涉及到反向扫描操作，暂不支持。
    - 如果有数据的话，RELATIVE 0重新抓取当前行。
  - count
    抓取随后的count行（和FORWARD count一样）。
  - ALL
    从当前关联位置开始，抓取所有剩余的行（和FORWARD ALL一样）。
  - FORWARD
    抓取下一行（和NEXT一样）。
  - FORWARD count
    与RELATIVE count的效果相同，从当前关联位置开始，抓取随后或前面的第count行。
  - FORWARD ALL
    从当前关联位置开始，抓取所有剩余行。
  - BACKWARD
    从当前关联位置开始，抓取前面一行(和PRIOR一样) 。
  - BACKWARD count
    从当前关联位置开始，抓取前面的count行（向后扫描）。
    取值范围：有符号的整数
    - count为正数就抓取当前关联位置之前的第count行。
    - count为负数就抓取当前关联位置之后的第abs（count）行。
    - 如果有数据的话，BACKWARD 0重新抓取当前行。
  - BACKWARD ALL
    从当前关联位置开始，抓取所有前面的行（向后扫描） 。
- **{ FROM | IN } cursor_name**
  使用关键字FROM或IN指定游标名称。
  取值范围：已创建的游标的名称。

示例

示例一：SELECT语句，用一个游标读取一个表。

建立一个名为cursor1的游标：

```
CURSOR cursor1 FOR SELECT * FROM customer_address ORDER BY 1;
```

抓取头3行到游标cursor1里。

```
FETCH FORWARD 3 FROM cursor1;
```

示例二：VALUES子句，用一个游标读取VALUES子句中的内容。

建立一个名为cursor2的游标：

```
CURSOR cursor2 FOR VALUES(1,2),(0,3) ORDER BY 1;
```

抓取头2行到游标cursor2里：


```
FETCH FORWARD 2 FROM cursor2;
```

相关链接

CLOSE，**MOVE**

12.83 MOVE

功能描述

MOVE在不检索数据的情况下重新定位一个游标。MOVE的作用类似于**FETCH**命令，但只是重定位游标而不返回行。

注意事项

无。

语法格式

```
MOVE [ direction [ FROM | IN ] ] cursor_name;
```

其中direction子句为可选参数。

```
NEXT
  | PRIOR
  | FIRST
  | LAST
  | ABSOLUTE count
  | RELATIVE count
  | count
  | ALL
  | FORWARD
  | FORWARD count
  | FORWARD ALL
  | BACKWARD
  | BACKWARD count
  | BACKWARD ALL
```

参数说明

MOVE命令的参数与FETCH的相同，详细请参见FETCH的**参数说明**。

说明

成功完成时，MOVE命令将返回一个“MOVE count”的标签，count是一个使用相同参数的FETCH命令会返回的行数（可能为零）。

示例

创建表reason，并向表中插入数据：

```
DROP TABLE IF EXISTS reason;
CREATE TABLE reason
(
  a    int primary key,
  b    int,
  c    int
);

INSERT INTO reason VALUES (1, 2, 3);
```

开始一个事务：

```
START TRANSACTION;
```

定义一个名为cursor1的游标：

```
CURSOR cursor1 FOR SELECT * FROM reason;
```

忽略游标cursor1的前3行：

```
MOVE FORWARD 3 FROM cursor1;
```

抓取游标cursor1的前4行：

```
FETCH 4 FROM cursor1;
```

关闭游标：

```
CLOSE cursor1;
```

结束一个事务：

```
END;
```

相关链接

CLOSE，**FETCH**

12.84 REINDEX

功能描述

为表中的数据重建索引。

在以下几种情况下需要使用REINDEX重建索引：

- 索引崩溃，并且不再包含有效的数据。
- 索引变得“臃肿”，包含大量的空页或接近空页。
- 为索引更改了存储参数（例如填充因子），并且希望这个更改完全生效。
  使用CONCURRENTLY选项创建索引失败，留下了一个“非法”索引。

注意事项

REINDEX DATABASE和SYSTEM这种形式的重建索引不能在事务块中执行。

语法格式

- 重建普通索引。
  ```
  REINDEX { INDEX |  TABLE | DATABASE | SYSTEM } name [ FORCE ];
  ```
- 重建索引分区。
  ```
  REINDEX  { TABLE } name
     PARTITION partition_name [ FORCE  ];
  ```

参数说明

- **INDEX**
  重新建立指定的索引。

- **TABLE**

  重新建立指定表的所有索引，如果表有从属的"TOAST"表，则这个表也会重建索引。

- **DATABASE**

  重建当前数据库里的所有索引。

- **SYSTEM**

  在当前数据库上重建所有系统表上的索引。不会处理在用户表上的索引。

- **name**

  需要重建索引的索引、表、数据库的名称。表和索引可以有模式修饰。

  说明

  REINDEX DATABASE和SYSTEM只能重建当前数据库的索引，所以name必须和当前数据库名称相同。

- **FORCE**

  无效选项，会被忽略。

- **partition_name**

  需要重建索引的分区的名字或者索引分区的名字。

  取值范围：

  - 如果前面是REINDEX INDEX，则这里应该指定索引分区的名字；
  - 如果前面是REINDEX TABLE，则这里应该指定分区的名字；

须知

REINDEX DATABASE和SYSTEM这种形式的重建索引不能在事务块中执行。

示例

重建分区表customer_address分区P1的分区索引：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);

CREATE INDEX customer_address_index on customer_address(CA_ADDRESS_SK) LOCAL;

REINDEX TABLE customer_address PARTITION P1;
```

12.85 RENAME TABLE

功能描述

重命名指定表。

注意事项

RENAME TABLE命令与如下命令的作用相同：

```
ALTER TABLE table_name RENAME to new_table_name
```

语法格式

```
RENAME TABLE
{[schema.]table_name TO new_table_name} [, ...];
```

参数说明

- **schema**

  模式名称。

- **table_name**

  需要修改的表名称。

- **new_table_name**

  修改后新的表名称。

示例

创建列存表指定存储格式和压缩方式：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH,COLVERSION=2.0)
DISTRIBUTE BY HASH (ca_address_sk);
```

修改表名：

```
RENAME TABLE customer_address TO new_customer_address;
```

相关链接

CREATE TABLE，**ALTER TABLE**，**DROP TABLE**

12.86 RESET

功能描述

RESET将指定的运行时参数恢复为缺省值。这些参数的缺省值是指postgresql.conf配置文件中所描述的参数缺省值。

RESET命令与如下命令的作用相同：

SET configuration_parameter TO DEFAULT

注意事项

RESET的事务性行为和SET相同：它的影响将会被事务回滚撤销。

语法格式

```
RESET {configuration_parameter | CURRENT_SCHEMA | TIME ZONE | TRANSACTION ISOLATION LEVEL | SESSION AUTHORIZATION | ALL };
```

参数说明

- **configuration_parameter**
  运行时参数的名称。
  取值范围：可以使用SHOW ALL命令查看运行时参数。

  说明

  部分通过SHOW ALL查看的参数不能通过SET设置。如max_datanodes。
- **CURRENT_SCHEMA**
  当前模式。
- **TIME ZONE**
  时区。
- **TRANSACTION ISOLATION LEVEL**
  事务的隔离级别。
- **SESSION AUTHORIZATION**
  当前会话的用户标识符。
- **ALL**
  所有运行时参数。

示例

把timezone设为缺省值：

```
RESET timezone;
```

把所有参数设置为缺省值：

```
RESET ALL;
```


相关链接

SET，SHOW

12.87 SET

功能描述

用于修改运行时配置参数。

注意事项

大多数运行时参数都可以用SET在运行时设置，但有些则在服务运行过程中或会话开始之后不能修改。

语法格式

- 设置所处的时区。
```
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT };
```
- 设置所属的模式。
```
SET [ SESSION | LOCAL ]
    {CURRENT_SCHEMA { TO | = } { schema | DEFAULT }
    | SCHEMA 'schema'};
```
- 设置客户端编码集。
```
SET [ SESSION | LOCAL ] NAMES encoding_name;
```
- 设置XML的解析方式。
```
SET [ SESSION | LOCAL ] XML OPTION { DOCUMENT | CONTENT };
```
- 设置其他运行时参数。
```
SET [ LOCAL | SESSION ]
    { {config_parameter { { TO | = } { value | DEFAULT }
                        | FROM CURRENT }}};
```

参数说明

- **SESSION**

  声明的参数只对当前会话起作用。如果SESSION和LOCAL都没出现，则SESSION为缺省值。

  如果在事务中执行了此命令，命令的产生影响将在事务回滚之后消失。如果该事务已提交，影响将持续到会话的结束，除非被另外一个SET命令重置参数。

- **LOCAL**

  声明的参数只在当前事务中有效。在COMMIT或ROLLBACK之后，会话级别的设置将再次生效。

  不论事务是否提交，此命令的影响只持续到当前事务结束。一个特例是：在一个事务里面，即有SET命令，又有SET LOCAL命令，且SET LOCAL在SET后面，则在事务结束之前，SET LOCAL命令会起作用，但事务提交之后，则是SET命令会生效。

- **TIME ZONE timezone**

  用于指定当前会话的本地时区。

  取值范围：有效的本地时区。该选项对应的运行时参数名称为TimeZone，DEFAULT缺省值为PRC。

- **CURRENT_SCHEMA**

  schema

  CURRENT_SCHEMA用于指定当前的模式。

  取值范围：已存在模式名称。

- **SCHEMA schema**

  同CURRENT_SCHEMA。此处的schema是个字符串。

  例如：set schema 'public';

- **NAMES encoding_name**

  用于设置客户端的字符编码。等价于set client_encoding to encoding_name。

  取值范围：有效的字符编码。该选项对应的运行时参数名称为client_encoding，默认编码为UTF8。

- **XML OPTION option**

  用于设置XML的解析方式。

  取值范围：CONTENT（缺省）、DOCUMENT

- **config_parameter**

  可设置的运行时参数的名称。可用的运行时参数可以使用SHOW ALL命令查看。

  说明

  部分通过SHOW ALL查看的参数不能通过SET设置。如max_datanodes。

- **value**

  config_parameter的新值。可以声明为字符串常量、标识符、数字，或者逗号分隔的列表。DEFAULT用于把这些参数设置为它们的缺省值。

示例

设置模式tpcds搜索路径：

```
SET search_path TO tpcds, public;
```

把日期时间风格设置为传统的 POSTGRES 风格(日在月前)：

```
SET datestyle TO postgres;
```

相关链接

RESET，**SHOW**

12.88 SET CONSTRAINTS

功能描述

SET CONSTRAINTS设置当前事务检查行为的约束条件。

IMMEDIATE约束是在每条语句后面进行检查。DEFERRED约束一直到事务提交时才检查。每个约束都有自己的模式。

从创建约束条件开始，一个约束总是设定为DEFERRABLE INITIALLY DEFERRED，DEFERRABLE INITIALLY IMMEDIATE，NOT DEFERRABLE三个特性之一。第三种总是

IMMEDIATE，并且不会受SET CONSTRAINTS影响。前两种以指定的方式启动每个事务，但是其行为可以在事务里用SET CONSTRAINTS改变。

带着一个约束名列表的SET CONSTRAINTS改变这些约束的模式（都必须是可推迟的）。如果有多个约束匹配某个名字，则所有都会被影响。SET CONSTRAINTS ALL改变所有可推迟约束的模式。

当SET CONSTRAINTS把一个约束从DEFERRED改成IMMEDIATE的时候，新模式反作用式地起作用：任何将在事务结束准备进行的数据修改都将在SET CONSTRAINTS的时候执行检查。如果违反了任何约束，SET CONSTRAINTS都会失败（并且不会修改约束模式）。因此，SET CONSTRAINTS可以用于强制在事务中某一点进行约束检查。

目前，只有外键约束被该设置影响。检查和唯一约束总是不可推迟的。

注意事项

SET CONSTRAINTS只在当前事务里设置约束的行为。因此，如果用户在事务块之外（START TRANSACTION/COMMIT对）执行这个命令，它将没有任何作用。

语法格式

```
SET CONSTRAINTS { ALL | { name } [, ...] } { DEFERRED | IMMEDIATE };
```

参数说明

- **name**

  约束名。

  取值范围：已存在的约束名。可以在系统表pg_constraint中查到。

- **ALL**

  所有约束。

- **DEFERRED**

  约束一直到事务提交时才检查。

- **IMMEDIATE**

  约束在每条语句后进行检查。

示例

设置所有约束在事务提交时检查：

```
SET CONSTRAINTS ALL DEFERRED;
```

12.89 SET ROLE

功能描述

设置当前会话的当前用户标识符。

注意事项

- 当前会话的用户必须是指定的rolename角色的成员，但系统管理员可以选择任何角色。

- 使用SET ROLE命令，可能会增加一个用户的权限，也可能会限制一个用户的权限。如果会话用户的角色有INHERITS属性，则自动拥有可以SET ROLE变成的角色的所有权限；在这种情况下，SET ROLE实际上是删除了所有直接赋予会话用户的权限，以及它的所属角色的权限，只剩下指定角色的权限。另一方面，如果会话用户的角色有NOINHERITS属性，SET ROLE删除直接赋予会话用户的权限，而获取指定角色的权限。

语法格式

- 设置当前会话的当前用户标识符。
  ```
  SET [ SESSION | LOCAL ] ROLE role_name PASSWORD 'password';
  ```
- 重置当前用户标识为当前会话用户标识符。
  ```
  RESET ROLE;
  ```

参数说明

- **SESSION**

  声明这个命令只对当前会话起作用，此参数为缺省值。

  取值范围：字符串，要符合标识符的命名规范。
- **LOCALE**

  声明该命令只在当前事务中有效。
- **role_name**

  角色名。

  取值范围：字符串，要符合标识符的命名规范。
- **password**

  角色的密码。要求符合密码的命名规则。
- **RESET ROLE**

  用于重置当前用户标识。

示例

创建一个角色，名为manager：

```
CREATE ROLE paul IDENTIFIED BY '{Password}';
```

设置当前用户为paul：

```
SET ROLE paul PASSWORD '{Password}';
```

查看当前会话用户，当前用户：

```
SELECT SESSION_USER, CURRENT_USER;
```

重置当前用户：

```
RESET role;
```

12.90 SET SESSION AUTHORIZATION

功能描述

把当前会话里的会话用户标识和当前用户标识都设置为指定的用户。

注意事项

只有在初始会话用户有系统管理员权限的时候，会话用户标识符才能改变。否则，只有在指定了被认证的用户名的情况下，系统才接受该命令。

语法格式

- 为当前会话设置会话用户标识符和当前用户标识符。
```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION role_name PASSWORD 'password';
```
- 重置会话和当前用户标识符为初始认证的用户名。
```
{SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
    | RESET SESSION AUTHORIZATION};
```

参数说明

- **SESSION**

  声明这个命令只对当前会话起作用。

  取值范围：字符串，要符合标识符的命名规范。
- **LOCALE**

  声明该命令只在当前事务中有效。
- **role_name**

  用户名。

  取值范围：字符串，要符合标识符的命名规范。
- **password**

  角色的密码。要求符合密码的命名规则。
- **DEFAULT**

  重置会话和当前用户标识符为初始认证的用户名。

示例

设置当前用户为paul：

```
SET SESSION AUTHORIZATION paul password '{Password}';
```

查看当前会话用户，当前用户：

```
SELECT SESSION_USER, CURRENT_USER;
```

重置当前用户：

```
RESET SESSION AUTHORIZATION;
```

相关参考

SET ROLE

12.91 SHOW

功能描述

SHOW将显示当前运行时参数的数值。可以使用SET语句来设置这些参数。

注意事项

SHOW可以查看的某些参数是只读的，但不能设置它们的值。

语法格式

```
SHOW
 {
   configuration_parameter |
   CURRENT_SCHEMA |
   TIME ZONE |
   TRANSACTION ISOLATION LEVEL |
   SESSION AUTHORIZATION |
   ALL
 };
```

参数说明

显示变量的参数请参见RESET的**参数说明**。

示例

显示 timezone参数值：

```
SHOW timezone;
```

显示参数DateStyle的当前设置：

```
SHOW DateStyle;
```

显示所有参数的当前设置：

```
SHOW ALL;
```

相关链接

SET，**RESET**

12.92 TRUNCATE

功能描述

清理表数据，TRUNCATE快速地从表中删除所有行。

它和在目标表上进行无条件的DELETE有同样的效果，但由于TRUNCATE不做表扫描，因而快得多。在大表上操作效果更明显。

注意事项

- TRUNCATE TABLE在功能上与不带WHERE子句DELETE语句相同：二者均删除表中的全部行。
- TRUNCATE TABLE比DELETE速度快且使用系统和事务日志资源少：
  - DELETE语句每次删除一行，并在事务日志中为所删除每行记录一项。
  - TRUNCATE TABLE通过释放存储表数据所用数据页来删除数据，并且只在事务日志中记录页的释放。
- TRUNCATE，DELETE，DROP三者的差异如下：

- TRUNCATE TABLE，删除内容，释放空间，但不删除定义。
- DELETE TABLE，删除内容，不删除定义，不释放空间。
- DROP TABLE，删除内容和定义，释放空间。

语法格式

- 清理表数据。

```
TRUNCATE [ TABLE ] [ ONLY ] {[[database_name.]schema_name.]table_name [ * ]} [, ... ]
    [ CONTINUE IDENTITY ] [ CASCADE | RESTRICT ];
```

- 清理表分区的数据。

```
ALTER TABLE [ IF EXISTS  ] { [ ONLY  ] [[database_name.]schema_name.]table_name
                        | table_name *
                        | ONLY ( table_name )  }
    TRUNCATE PARTITION { partition_name
                    | FOR (  partition_value  [, ...] )  } ;
```

参数说明

- **ONLY**

  如果声明ONLY，只有指定的表会被清空。如果没有声明ONLY，这个表以及其所有子表（若有）会被清空。

- **database_name**

  目标表的数据库名称。

- **schema_name**

  目标表的模式名称。

- **table_name**

  目标表的名字（可以有模式修饰）。

  取值范围：已存在的表名。

- **CONTINUE IDENTITY**

  不改变序列的值。这是缺省值。

- **CASCADE | RESTRICT**
  - CASCADE：级联清空所有在该表上有外键引用的表，或者由于CASCADE而被添加到组中的表。
  - RESTRICT（缺省值）：如果其他表在该表上有外键引用则拒绝清空。

- **partition_name**

  目标分区表的分区名。

  取值范围：已存在的分区名。

- **partition_value**

  指定的分区键值。

  通过PARTITION FOR子句指定的这一组值，可以唯一确定一个分区。

  取值范围：需要进行删除数据分区的分区键的取值范围。

须知

使用PARTITION FOR子句时，partition_value所在的整个分区会被清空。

示例

创建分区表customer_address：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
```

清空表customer_address分区p1：

```
ALTER TABLE customer_address TRUNCATE PARTITION p1;
```

清空分区表：

```
TRUNCATE TABLE customer_address;
```

12.93 VACUUM

功能描述

VACUUM回收表或B-Tree索引中已经删除的行所占据的存储空间。在一般的数据库操作里，那些已经DELETE的行并没有从它们所属的表中物理删除；在完成VACUUM之前它们仍然存在。因此有必要周期地运行VACUUM，特别是在经常更新的表上。

注意事项

- 如果没有参数，VACUUM处理当前数据库里用户拥有相应权限的每个表。如果参数指定了一个表，VACUUM只处理指定的那个表。
- 要对一个表进行VACUUM操作，通常用户必须是表的所有者，被授予了指定表VACUUM权限的用户或者被授予了gs_role_vacuum_any角色的用户，系统管理员默认拥有此权限。数据库的所有者允许对数据库中除了共享目录以外的所有表进行VACUUM操作（该限制意味着只有系统管理员才能真正对一个数据库进行VACUUM操作）。VACUUM命令会跳过那些用户没有权限的表进行垃圾回收操作。
- VACUUM不能在事务块内执行。
- 建议生产数据库经常清理（至少每晚一次），以保证不断地删除失效的行。尤其是在增删了大量记录之后，对受影响的表执行VACUUM ANALYZE命令是一个很好的习惯。这样将更新系统目录为最近的更改，并且允许查询优化器在规划用户查询时有更好的选择。
- 不建议日常使用FULL选项，但是可以在特殊情况下使用。例如在用户删除了一个表的大部分行之后，希望从物理上缩小该表以减少磁盘空间占用。VACUUM FULL通常要比单纯的VACUUM收缩更多的表尺寸。如果执行此命令后所占用物理

空间无变化（未减少），请确认是否有其他活跃事务（删除数据事务开始之前开始的事务，并在VACUUM FULL执行前未结束）存在，如果有等其他活跃事务退出进行重试。

- VACUUM会导致I/O流量的大幅增加，这可能会影响其他活动会话的性能。因此，有时候会建议使用基于开销的VACUUM延迟特性。
- 如果指定了VERBOSE选项，VACUUM将打印处理过程中的信息，以表明当前正在处理的表。各种有关当前表的统计信息也会打印出来。
- 语法格式中含有带括号的选项列表时，选项可以通过任何顺序写入。如果没有括号，则选项必须按语法显示的顺序给出。
- VACUUM和VACUUM FULL时，会根据参数vacuum_defer_cleanup_age延迟清理行存表记录，即不会立即清理刚刚删除的元组。
- VACUUM ANALYZE先执行一个VACUUM操作，然后给每个选定的表执行一个ANALYZE。对于日常维护脚本而言，这是一个很方便的组合。
- 简单的VACUUM（不带FULL选项）只是简单地回收空间并且令其可以再次使用。这种形式的命令可以和对表的普通读写并发操作，因为没有请求排他锁。VACUUM FULL执行更广泛的处理，包括跨块移动行，以便把表压缩到最少的磁盘块数目里。这种形式要慢许多并且在处理的时候需要在表上施加一个排他锁。
- VACUUM列存表内部执行的操作包括三个：迁移delta表中的数据到主表、VACUUM主表的delta表、VACUUM主表的desc表。该操作不会回收delta表的存储空间，如果要回收delta表的冗余存储空间，需要对该列存表执行VACUUM DELTAMERGE。
- 如果有长查询访问系统表，此时执行VACUUM FULL，长查询可能会阻塞VACUUM FULL连接访问系统表，导致连接超时报错。
- 对列存分区表执行VACUUM FULL，会同时锁表和锁分区。
- 并发VACUUM FULL系统表可能会导致本地死锁。
- 对表执行VACUUM FULL操作时会触发表重建（表重建过程中会先把数据转储到一个新的数据文件中，重建完成之后会删除原始文件），当表比较大时，重建会消耗较多的磁盘空间。当磁盘空间不足时，要谨慎对待大表VACUUM FULL操作，防止触发集群只读。

语法格式

- 回收空间并更新统计信息，关键字顺序必须按语法显示的顺序给出。
```
VACUUM [ ( { FULL | FREEZE | VERBOSE | {ANALYZE | ANALYSE }} [,...] ) ]
    [ table_name [ (column_name [, ...] ) ] ] [ PARTITION ( partition_name ) ];
```
- 仅回收空间，不更新统计信息。
```
VACUUM [ FULL [COMPACT] ] [ FREEZE ] [ VERBOSE ] [ table_name ] [ PARTITION
( partition_name ) ];
```
- 回收空间并更新统计信息，且对关键字顺序有要求。
```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] { ANALYZE | ANALYSE } [ VERBOSE ]
    [ table_name [ (column_name [, ...] ) ] ] [ PARTITION ( partition_name ) ];
```
- 针对HDFS表，将delta table中的数据转移到主表存储。
```
VACUUM DELTAMERGE [ table_name ];
```
- 针对HDFS表，删除HDFS表在HDFS存储上的空值分区目录。
```
VACUUM HDFSDIRECTORY [ table_name ];
```

参数说明

- **FULL**

选择“FULL”清理，这样可以恢复更多的空间，但是需要耗时更多，并且在表上施加了排他锁。

FULL选项还可以带有COMPACT参数，该参数只针对HDFS表，指定该参数的VACUUM FULL操作性能要好于未指定该参数的VACUUM FULL操作。

COMPACT和PARTITION参数不能同时使用。

说明

使用FULL参数会导致统计信息丢失，如果需要收集统计信息，请在VACUUM FULL语句中加上analyze关键字。

- **FREEZE**

  指定FREEZE相当于执行VACUUM时将GUC参数**vacuum_freeze_min_age**设为0。

- **VERBOSE**

  为每个表打印一份详细的清理工作报告。

- **ANALYZE | ANALYSE**

  更新用于优化器的统计信息，以决定执行查询的最有效方法。

- **table_name**

  要清理的表的名称（可以有模式修饰）。

  取值范围：要清理的表的名称。缺省时为当前数据库中的所有表。

- **column_name**

  要分析的具体的字段名称。

  取值范围：要分析的具体的字段名称。缺省时为所有字段。

- **PARTITION**

  HDFS表不支持PARTITION参数，PARTITION参数不能和COMPACT同时使用。

  说明

  PARTITION参数和COMPACT同时使用会报错：COMPACT can not be used with PARTITION.

- **partition_name**

  要清理的表的分区名称。缺省时为所有分区。

- **DELTAMERGE**

  只针对HDFS表，将HDFS表的delta table中的数据转移到主表存储上。对HDFS表而言，当delta表中数据量小于六万行，则不作迁移，只有在大于或者等于六万行数据时，将delta表中所有数据迁移到HDFS上，并通过truncate清理delta表的存储空间。

- **HDFSDIRECTORY**

  只针对HDFS表，删除HDFS表在HDFS存储上表目录下的空值分区目录。

示例

创建分区表customer_address：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER               NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
```

```
    ca_street_name      CHARACTER varying(60)               ,
    ca_street_type      CHARACTER(15)                       ,
    ca_suite_number     CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
```

清理当前数据库中的所有表：

```
VACUUM;
```

仅回收表customer_address分区P2的空间，不更新统计信息：

```
VACUUM FULL customer_address PARTITION(P2);
```

回收表customer_address空间，并更新统计信息：

```
VACUUM FULL ANALYZE customer_address;
```

清理当前数据库中的所有表并收集查询优化器的统计信息：

```
VACUUM ANALYZE;
```

仅清理特定表reason：

```
VACUUM (VERBOSE, ANALYZE) customer_address;
```


13 DML 语法

13.1 DML 语法一览表

DML（Data Manipulation Language数据操作语言），用于对数据库表中的数据进行操作。如：插入、更新、查询、删除。

插入数据

插入数据是往数据库表中添加一条或多条记录，请参考**INSERT**。

修改数据

修改数据是修改数据库表中的一条或多条记录，请参考**UPDATE**。

查询数据

数据库查询语句SELECT是用于在数据库中检索适合条件的信息，请参考**SELECT**。

删除数据

删除表中指定条件的数据，请参考**DELETE**。

复制数据

GaussDB(DWS)提供了在表和文件之间复制数据的语句，请参考**COPY**。

锁定表

GaussDB(DWS)提供了多种锁模式用于控制对表中数据的并发访问，请参考**LOCK**。

调用函数

GaussDB(DWS)提供了三个用于调用函数的语句，它们在语法结构上没有差别，请参考**CALL**。

13.2 CALL

功能描述

使用CALL命令可以调用已定义的函数和存储过程。

注意事项

如果自定义函数的函数名与系统函数同名，则在调用自定义函数时需指定Schema，否则系统会优先调用系统函数。

语法格式

```
CALL [schema.] {func_name| procedure_name} ( param_expr );
```

参数说明

- **schema**

  函数或存储过程所在的模式名称。

- **func_name**

  所调用函数或存储过程的名称。

  取值范围：已存在的函数名称。

- **param_expr**

  参数列表可以用符号“:=”或者“=>”将参数名和参数值隔开，这种方法的好处是参数可以以任意顺序排列。若参数列表中仅出现参数值，则参数值的排列顺序必须和函数或存储过程定义时的相同。

  取值范围：已存在的函数参数名称或存储过程参数名称。

  说明

  参数可以包含入参（参数名和类型之间指定“IN”关键字）和出参（参数名和类型之间指定“OUT”关键字），使用CALL命令调用函数或存储过程时，对于非重载的函数，参数列表必须包含出参，出参可以传入一个变量或者任一常量，详见**示例**。对于重载的package函数，参数列表里可以忽略出参，忽略出参时可能会导致函数找不到。包含出参时，出参只能是常量。

示例

创建一个函数func_add_sql，计算两个整数的和，并返回结果：

```
CREATE FUNCTION func_add_sql(num1 integer, num2 integer) RETURN integer
AS
BEGIN
RETURN num1 + num2;
END;
/
```

按参数值传递：

```
CALL func_add_sql(1, 3);
```

使用命名标记法传参：

```
CALL func_add_sql(num1 => 1,num2 => 3);
CALL func_add_sql(num2 := 2, num1 := 3);
```

删除函数：

```
DROP FUNCTION func_add_sql;
```

创建带出参的函数：

```
CREATE FUNCTION func_increment_sql(num1 IN integer, num2 IN integer, res OUT integer)
RETURN integer
AS
BEGIN
res := num1 + num2;
END;
/
```

出参传入常量：

```
CALL func_increment_sql(1,2,1);
```

出参传入变量：

```
DECLARE
res int;
BEGIN
func_increment_sql(1, 2, res);
dbms_output.put_line(res);
END;
/
```

创建重载的函数：

```
create or replace procedure package_func_overload(col int, col2 out int) package
as
declare
    col_type text;
begin
     col := 122;
         dbms_output.put_line('two out parameters ' || col2);
end;
/
create or replace procedure package_func_overload(col int, col2 out varchar)
package
as
declare
    col_type text;
begin
     col2 := '122';
         dbms_output.put_line('two varchar parameters ' || col2);
end;
/
```

函数调用：

```
call package_func_overload(1, 'test');
call package_func_overload(1, 1);
```

删除函数：

```
DROP FUNCTION func_increment_sql;
```

13.3 COPY

功能描述

通过COPY命令实现在表和文件之间复制数据。

COPY FROM从一个文件复制数据到一个表，COPY TO把一个表的数据复制到一个文件。

注意事项

- 以安全模式(云上安全模式不支持关闭)启动CN、DN的开关，那么当前模式下禁止使用COPY FROM FILENAME或COPY TO FILENAME语法，可采用\copy的方式进行规避，请参考**如何使用\copy导入导出**中的示例。
- COPY只能用于表，不能用于视图。
- 对任何要插入数据的表必须有插入权限。
- 如果声明了一个字段列表，COPY将只在文件和表之间复制已声明字段的数据。如果表中有任何不在字段列表里的字段，COPY FROM将为那些字段插入缺省值。
- 如果声明了数据源文件，服务器必须可以访问该文件；如果指定了STDIN，数据将在客户前端和服务器之间流动，输入时，表的列与列之间使用TAB键分隔，在新的一行中以反斜杠和句点（\.）表示输入结束。
- 如果数据文件的任意行包含比预期多或者少的字段，COPY FROM将抛出一个错误。
- 数据的结束可以用一个只包含反斜杠和句点（\.）的行表示。如果从文件中读取数据，数据结束的标记是不必要的；如果在客户端应用之间复制数据，必须要有结束标记。
- COPY FROM中\N为空字符串，如果要输入实际数据值\N ，使用\\N。
- COPY FROM不支持在导入过程中对数据做预处理（比如说表达式运算，填充指定默认值等）。如果需要在导入过程中对数据做预处理，用户需先把数据导入到临时表中，然后执行SQL语句通过运算插入到表中，但此方法会导致I/O膨胀，降低导入性能。
- COPY FROM在遇到数据格式错误时会回滚事务，但没有足够的错误信息，不方便用户从大量的原始数据中定位错误数据。
- COPY FROM/TO适合低并发，本地小数据量导入导出。
- 支持使用COPY TO向OBS导出TEXT或CSV格式数据，但不支持使用COPY FROM从OBS导入数据。对于导出到OBS上的带utf8 BOM的CSV格式文件，如果有导入需求，建议使用OBS导入，否则可能无法正确识别BOM字段。

语法格式

- 从一个文件复制数据到一个表：

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ USING ] DELIMITERS 'delimiters' ]
    [ WITHOUT ESCAPING ]
    [ LOG ERRORS ]
    [ LOG ERRORS data ]
    [ REJECT LIMIT 'limit' ]
    [ [ WITH ] ( option [, ...] ) ]
    | copy_option
    | FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[  ...] ] ];
```

  说明

  语法中的FIXED FORMATTER ({ column_name(offset, length) } [, ...])以及 [(option [, ...]) | copy_option [...]] 可以任意排列组合。

- 把一个表的数据复制到一个文件：

```
COPY table_name [ ( column_name [, ...] ) ]
    TO { 'filename' | STDOUT }
    [ [ USING ] DELIMITERS 'delimiters' ]
    [ WITHOUT ESCAPING ]
    [ [ WITH ] ( option [, ...] ) ]
```

```
   | copy_option
   | FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[  ...] ] ];

COPY query
   TO { 'filename' | STDOUT }
   [ WITHOUT ESCAPING ]
   [ [ WITH ] ( option [, ...] ) ]
   | copy_option
   | FIXED FORMATTER ( { column_name( offset, length ) } [, ...] ) [ ( option [, ...] ) | copy_option
[  ...] ] ];
```

说明

1. COPY TO语法形式约束如下：

   (query)与[USING] DELIMITER不兼容，即若COPY TO的数据来自于一个query的查询结果，那么COPY TO语法不能再指定[USING] DELIMITERS语法子句。

2. 对于FIXED FORMATTTER语法后面跟随的copy_option是以空格进行分隔的。
3. copy_option是指COPY原生的参数形式，而option是兼容外表导入的参数形式。
4. 语法中的FIXED FORMATTER ({ column_name(offset, length) } [, ...])以及[(option [, ...]) | copy_option [...]] 可以任意排列组合。

其中可选参数option子句语法为：

```
FORMAT 'format_name'
| OIDS [ boolean ]
| DELIMITER 'delimiter_character'
| NULL 'null_string'
| HEADER [ boolean ]
| FILEHEADER 'header_file_string'
| FREEZE [ boolean ]
| QUOTE 'quote_character'
| ESCAPE 'escape_character'
| EOL 'newline_character'
| NOESCAPING [ boolean ]
| FORCE_QUOTE { ( column_name [, ...] ) | * }
| FORCE_NOT_NULL ( column_name [, ...] )
| ENCODING 'encoding_name'
| IGNORE_EXTRA_DATA [ boolean ]
| FILL_MISSING_FIELDS [ boolean ]
| COMPATIBLE_ILLEGAL_CHARS [ boolean ]
| DATE_FORMAT 'date_format_string'
| TIME_FORMAT 'time_format_string'
| TIMESTAMP_FORMAT 'timestamp_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
| SERVER 'obs_server_string'
| BOM [ boolean ]
| MAXROW [ integer ]
| FILEPREFIX 'file_prefix_string'
```

其中可选参数copy_option子句语法为：

```
OIDS
| NULL 'null_string'
| HEADER
| FILEHEADER 'header_file_string'
| FREEZE
| FORCE_NOT_NULL column_name [, ...]
| FORCE_QUOTE { column_name [, ...] | * }
| BINARY
| CSV
| QUOTE [ AS ] 'quote_character'
| ESCAPE [ AS ] 'escape_character'
| EOL 'newline_character'
| ENCODING 'encoding_name'
| IGNORE_EXTRA_DATA
| FILL_MISSING_FIELDS
| COMPATIBLE_ILLEGAL_CHARS
| DATE_FORMAT 'date_format_string'
| TIME_FORMAT 'time_format_string'
```



```
| TIMESTAMP_FORMAT 'timestamp_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
```

参数说明

- **query**

  其结果将被复制。

  取值范围：一个必须用圆括弧包围的SELECT或VALUES命令。

- **table_name**

  表的名字（可以有模式修饰）。

  取值范围：已存在的表名。

- **column_name**

  可选的待复制字段列表。

  取值范围：如果没有声明字段列表，将使用所有字段。

- **STDIN**

  声明输入是来自标准输入。

- **STDOUT**

  声明输出打印到标准输出。

- **FIXED**

  打开字段固定长度模式。在字段固定长度模式下，不能声明DELIMITER，NULL，CSV选项。指定FIXED类型后，不能再通过option或copy_option指定BINARY、CSV、TEXT等类型。

  说明

  定长格式定义如下：

  1. 每条记录的每个字段长度相同。
  2. 长度不足的字段以空格填充，数字类型字段左对齐，字符字段右对齐。
  3. 字段和字段之间没有分隔符。

- **[USING] DELIMITER 'delimiters'**

  在文件中分隔各个字段的字符串，分隔符最大长度不超过10个字节。

  取值范围：不允许包含\.abcdefghijklmnopqrstuvwxyz0123456789中的任何一个字符。

  缺省值：在文本模式下，缺省是水平制表符，在CSV模式下是一个逗号。

- **WITHOUT ESCAPING**

  在TEXT格式中，不对'\'和后面的字符进行转义。

  取值范围：仅支持TEXT格式。

- **LOG ERRORS**

  若指定，则开启对于COPY FROM语句中数据类型错误的容错机制，相关错误行的错误记录会记录到此库中public.pgxc_copy_error_log表中，备后续查阅。

  取值范围：仅支持导入（即COPY FROM）时指定。

说明

此容错选项的使用限制如下：

- 此容错机制仅捕捉COPY FROM过程中CN节点上数据解析过程中相关的数据类型错误（DATA_EXCEPTION），诸如CN与DN之间的网络交互错误或者是DN上的表达式转换错误等CN数据解析逻辑之外的过程无法涵盖在内。
- 在每个库第一次使用时COPY FROM容错时，请先行检查public.pgxc_copy_error_log（COPY错误表）是否存在，若不存在请调用copy_error_log_create() 函数创建；若存在，请转移此表数据并删除这张表后，调用copy_error_log_create() 函数创建。更多关于表public.pgxc_copy_error_log的字段信息，请参见**表6-21**。
- 若在指定了LOG ERRORS的COPY FROM运行时，public.pgxc_copy_error_log不存在（未创建或者已删除）或表定义不符合copy_error_log_create() 中的预设表定义，则会报错。因此请确定此COPY错误表是使用copy_error_log_create() 函数创建的，否则可能导致容错的COPY FROM语句无法正常执行。
- COPY已有的容错选项（如IGNORE_EXTRA_DATA）开启时，对应类型的错误会按照已有的方式处理而不会报出异常，因此错误表也不会有相应数据。
- 此容错机制的覆盖范围与GDS Foreign Table的容错范围相同。推荐用户通过表名列以及COPY FROM语句开始时间戳对查询结果进行过滤。错误数据处理可参考处理错误表章节。

- **LOG ERRORS DATA**

  LOG ERRORS DATA和LOG ERRORS的区别：

  a. LOG ERRORS DATA会填充容错表的rawrecord字段。

  b. 只有supper权限的用户才能使用LOG ERRORS DATA参数选项。

  注意

  使用“**LOG ERRORS DATA**”时，若错误内容过于复杂可能存在写入容错表失败的风险，导致任务失败。

- **REJECT LIMIT 'limit'**

  与LOG ERROR选项共同使用，对COPY FROM的容错机制设置数值上限，一旦此COPY FROM语句错误数据超过选项指定条数，则会按照原有机制报错。

  取值范围：正整数（1-INTMAX），'unlimited'（无最大值限制）

  缺省值：若未指定LOG ERRORS，则会报错；若指定LOG ERRORS，则默认为0。

  说明

  如上述LOG ERRORS中描述的容错机制，REJECT LIMIT的计数也是按照执行COPY FROM的CN上遇到的解析错误数量计算，而不是每个DN上的错误数量，这点请与GDS容错机制区别开。

- **FORMATTER**

  在固定长度模式中，定义每一个字段在数据文件中的位置。按照column(offset,length)格式定义每一列在数据文件中的位置。

  取值范围：

  - offset取值不能小于0，以字节为单位。
  - length取值不能小于0，以字节为单位。

  所有列的总长度和不能大于1GB。

  文件中没有出现的列默认以空值代替。

- **OPTION { option_name ' value ' }**

  用于指定兼容外表的各类参数。

  - FORMAT

    数据源文件的格式。

    取值范围：CSV、TEXT、FIXED、BINARY。

    - CSV格式的文件，可以有效处理数据列中的换行符，但对一些特殊字符处理有欠缺。
    - TEXT格式的文件，可以有效处理一些特殊字符，但无法正确处理数据列中的换行符。
    - FIXED格式的文件，适用于每条数据的数据列都比较固定的数据，长度不足的列会添加空格补齐，过长的列则会自动截断。
    - BINARY形式的选项会使得所有的数据被存储/读作二进制格式而不是文本。 这比TEXT和CSV格式的要快一些，但是一个BINARY格式文件可移植性比较差。

    缺省值：TEXT

  - OIDS

    为每行复制内部对象标识（oid）。

    说明

    若COPY FROM对象为query或者对于没有oid的表，指定oids标识报错。

    取值范围：true/on，false/off。

    缺省值：false

  - DELIMITER

    指定数据文件行数据的字段分隔符。

    说明

    - 分隔符不能是\r和\n。
    - 分隔符不能和null参数相同，CSV格式数据的分隔符不能和quote参数相同。
    - TEXT格式数据的分隔符不能包含： 小写字母、数字和特殊字符.\。
    - 数据文件中单行数据长度需<1GB，如果分隔符较长且数据列较多的情况下，会影响导出有效数据的长度。
    - 分隔符推荐使用多字符和不可见字符。多字符例如'$^&'；不可见字符例如0x07，0x08，0x1b等。
    - 建议TEXT格式下多字符分隔符中的字符不要完全相同，例如不建议使用 delimiter '---' 。

    取值范围：支持多字符分隔符，但分隔符不能超过10个字节。

    缺省值：

    - TEXT格式的默认分隔符是水平制表符（tab）。
    - CSV格式的默认分隔符为“,”。
    - FIXED格式没有分隔符。

  - NULL

    用来指定数据文件中空值的表示。

取值范围：

- null值不能是\r和\n，最大为100个字符。
- null值不能和分隔符、quote参数相同。

缺省值：

- CSV格式下默认值是一个没有引号的空字符串。
- 在TEXT格式下默认值是\N。

– HEADER

指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。header只能用于CSV，FIXED格式的文件中。

在导入数据时，如果header选项为on，则数据文本第一行会被识别为标题行，会忽略此行。如果header为off，而数据文件中第一行会被识别为数据。

在导出数据时，如果header选项为on，则需要指定fileheader。如果header为off，则导出数据文件不包含标题行。

取值范围：true/on，false/off。

缺省值：false

– QUOTE

CSV格式文件下，指定一个数据值被引用时使用的引用字符。

缺省值：双引号

说明

- quote参数不能和分隔符、null参数相同。
- quote参数只能是单字节的字符。
- 推荐不可见字符作为quote，例如0x07，0x08，0x1b等。

– ESCAPE

CSV格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。

缺省值：和QUOTE相同。

– EOL 'newline_character'

指定导入导出数据文件换行符样式。

取值范围：支持多字符换行符，但换行符不能超过10个字节。常见的换行符，如\r、\n、\r\n（设成0x0D、0x0A、0x0D0A效果是相同的），其他字符或字符串，如$、#。

说明

- EOL参数只能用于TEXT格式的导入导出，不支持CSV格式和FIXED格式导入。为了兼容原有EOL参数，仍然支持导出CSV格式和FIXED格式时指定EOL参数为0x0D或0x0D0A。
- EOL参数不能和分隔符、null参数相同。
- EOL参数不能包含：.abcdefghijklmnopqrstuvwxyz0123456789。

– FORCE_QUOTE { (column_name [, ...]) | * }

在CSV COPY TO模式下，强制对每个声明的字段的所有非NULL值都使用引号包围。NULL输出不会被引号包围。

取值范围：已存在的字段。

- FORCE_NOT_NULL (column_name [, ...])

  在CSV COPY FROM模式下，指定的字段输入不能为空。

  取值范围：已存在的字段。

- ENCODING

  指定数据文件的编码格式名称，缺省为当前数据库编码格式。

- IGNORE_EXTRA_DATA

  若数据源文件比外表定义列数多，是否会忽略对多出的列。该参数只在数据导入过程中使用。

  取值范围：true/on、false/off。

  - 参数为true/on，若数据源文件比外表定义列数多，则忽略行尾多出来的列。
  - 参数为false/off，若数据源文件比外表定义列数多，会显示如下错误信息。

    ```
    extra data after last expected column
    ```

  缺省值：false/off

  须知

  如果行尾换行符丢失，使两行变成一行时，设置此参数为true将导致后一行数据被忽略掉。

- COMPATIBLE_ILLEGAL_CHARS

  导入非法字符容错参数。此语法仅对COPY FROM导入有效。

  取值范围：true/on，false/off。

  - 参数为true/on，则导入时遇到非法字符进行容错处理，非法字符转换后入库，不报错，不中断导入。
  - 参数为false/off，导入时遇到非法字符进行报错，中断导入。

  缺省值：false/off

  说明

  导入非法字符容错规则如下：

  （1）对于'\0'，容错后转换为空格；

  （2）对于其他非法字符，容错后转换为问号；

  （3）若compatible_illegal_chars为true/on标识导入时对于非法字符进行容错处理，则若NULL、DELIMITER、QUOTE、ESCAPE设置为空格或问号则会通过如"illegal chars conversion may confuse COPY escape 0x20"等报错信息提示用户修改可能引起混淆的参数以避免导入错误。

- FILL_MISSING_FIELDS

  当数据加载时，若数据源文件中一行的最后一个字段缺失的处理方式。

  取值范围：true/on，false/off。

  - 参数为true/on，当数据导入时，若数据源文件中一行数据的最后一个字段缺失，则把最后一个字段的值设置为NULL，不报错。

- 参数为false/off，如果最后一个字段缺失会显示如下错误信息。
  ```
  missing data for column "tt"
  ```

  缺省值：false/off

- DATE_FORMAT

  导入对于DATE类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法DATE格式。可参考**时间、日期处理函数和操作符**。

  说明

  对于指定为ORACLE兼容类型的数据库，则DATE类型内建为TIMESTAMP类型。在导入的时候，若需指定格式，可以参考下面的timestamp_format参数。

- TIME_FORMAT

  导入对于TIME类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法TIME格式，不支持时区。可参考**时间、日期处理函数和操作符**。

- TIMESTAMP_FORMAT

  导入对于TIMESTAMP类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法TIMESTAMP格式，不支持时区。可参考**时间、日期处理函数和操作符**。

- SMALLDATETIME_FORMAT

  导入对于SMALLDATETIME类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法SMALLDATETIME格式。可参考**时间、日期处理函数和操作符**。

- SERVER

  指定OBS SERVER，指定该参数时filename为OBS上的路径，表示导出到OBS。

  取值范围：已创建的OBS SERVER名。

  说明

  仅在COPY TO时生效。

- BOM

  标识是否对导出的CSV格式文件添加utf8 BOM字段。

  取值范围：true/on，false/off。

  缺省值：false

  说明

  仅在COPY TO并且指定了有效SERVER参数时生效，仅支持导出文件为utf8编码。

- MAXROW

  标识导出文件的最大行数，超出该值会生成新的文件。

取值范围：1~ 2147483647。

说明

仅在COPY TO并且指定了有效SERVER参数时生效。当HEADER取值为true时MAXROW需要大于1。该参数需要与FILEPREFIX同时指定。

- FILEPREFIX

  指定导出文件名的前缀。

  取值范围：合法的字符串，且不能以/开始或结尾。

  说明

  仅在COPY TO并且指定了有效SERVER参数时生效。该参数需要与MAXROW同时指定。

- **COPY_OPTION { option_name ' value ' }**

  用于指定COPY原生的各类参数。

  - OIDS

    为每行复制内部对象标识（oid）。

    说明

    若COPY FROM对象为query或者对于没有oid的表，指定oids标识报错。

  - NULL null_string

    用来指定数据文件中空值的表示。

    须知

    在使用COPY FROM的时候，任何匹配这个字符串的字符串将被存储为NULL值，所以应该确保指定的字符串和COPY TO相同。

    取值范围：

    - null值不能是\r和\n，最大为100个字符。
    - null值不能和分隔符、quote参数相同。

    缺省值：

    - 在TEXT格式下默认值是\N。
    - CSV格式下默认值是一个没有引号的空字符串。

  - HEADER

    指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。header只能用于CSV，FIXED格式的文件中。

    在导入数据时，如果header选项为on，则数据文本第一行会被识别为标题行，会忽略此行。如果header为off，而数据文件中第一行会被识别为数据。

    在导出数据时，如果header选项为on，则需要指定fileheader。如果header为off，则导出数据文件不包含标题行。

  - FILEHEADER

    导出数据时用于定义标题行的文件，一般用来描述每一列的数据信息。

须知

- 仅在header为on或true的情况下有效。
- fileheader指定的是绝对路径。
- 该文件只能包含一行标题信息，并以换行符结尾，多余的行将被丢弃（标题信息不能包含换行符）。
- 该文件包括换行符在内长度不超过1M。

- FREEZE

  将COPY加载的数据行设置为已经被frozen，就像这些数据行执行过VACUUM FREEZE。

  这是一个初始数据加载的性能选项。仅当以下三个条件同时满足时，数据行会被frozen：

  - 在同一事务中create或truncate这张表之后执行COPY。
  - 当前事务中没有打开的游标。
  - 当前事务中没有原有的快照。

  说明

  COPY完成后，所有其他会话将会立刻看到这些数据。但是这违反了MVCC可见性的一般原则，用户应当了解这样会导致潜在的风险。

- FORCE NOT NULL column_name [, ...]

  在CSV COPY FROM模式下，指定的字段输入不能为空。

  取值范围：已存在的字段。

- FORCE QUOTE { column_name [, ...] | * }

  在CSV COPY TO模式下，强制在每个声明的字段周围对所有非NULL值都使用引号包围。NULL输出不会被引号包围。

  取值范围：已存在的字段。

- BINARY

  使用二进制格式存储和读取，而不是以文本的方式。在二进制模式下，不能声明DELIMITER，NULL，CSV选项。指定BINARY类型后，不能再通过option或copy_option指定CSV、FIXED、TEXT等类型。

- CSV

  打开逗号分隔变量（CSV）模式。指定CSV类型后，不能再通过option或copy_option指定BINARY、FIXED、TEXT等类型。

- QUOTE [AS] 'quote_character'

  CSV格式文件下的引号字符。

  缺省值：双引号。

  说明

  - quote参数不能和分隔符、null参数相同。
  - quote参数只能是单字节的字符。
  - 推荐不可见字符作为quote，例如0x07，0x08，0x1b等。

- ESCAPE [AS] 'escape_character'

CSV格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。

默认值为双引号。当与quote值相同时，会被替换为'\0'。

- EOL 'newline_character'

  指定导入导出数据文件换行符样式。

  取值范围：支持多字符换行符，但换行符不能超过10个字节。常见的换行符，如\r、\n、\r\n（设成0x0D、0x0A、0x0D0A效果是相同的），其他字符或字符串，如$、#。

  说明

  - EOL参数只能用于TEXT格式的导入导出，不支持CSV格式和FIXED格式。为了兼容原有EOL参数，仍然支持导出CSV格式和FIXED格式时指定EOL参数为0x0D或0x0D0A。
  - EOL参数不能和分隔符、null参数相同。
  - EOL参数不能包含：.abcdefghijklmnopqrstuvwxyz0123456789。

- ENCODING 'encoding_name'

  指定文件编码格式名称。

  取值范围：有效的编码格式。

  缺省值：当前编码格式。

- IGNORE_EXTRA_DATA

  指定当数据源文件比外表定义列数多时，忽略行尾多出来的列。该参数只在数据导入过程中使用。

  若不使用该参数，在数据源文件比外表定义列数多，会显示如下错误信息。

  ```
  extra data after last expected column
  ```

- COMPATIBLE_ILLEGAL_CHARS

  指定导入时对非法字符进行容错处理，非法字符转换后入库。不报错，不中断导入。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  若不使用该参数，导入时遇到非法字符进行报错，中断导入。

  说明

  导入非法字符容错规则如下：

  1. 对于'\0'，容错后转换为空格。
  2. 对于其他非法字符，容错后转换为问号。
  3. 如果compatible_illegal_chars为true/on标识，导入时对于非法字符进行容错处理，则若NULL、DELIMITER、QUOTE、ESCAPE设置为空格或问号则会通过如"illegal chars conversion may confuse COPY escape 0x20"等报错信息提示用户修改可能引起混淆的参数以避免导入错误。

- FILL_MISSING_FIELDS

  当数据加载时，若数据源文件中一行的最后一个字段缺失的处理方式。

  取值范围：true/on，false/off。

  - 参数为true/on，当数据导入时，若数据源文件中一行数据的最后一个字段缺失，则把最后一个字段的值设置为NULL，不报错。
  - 参数为false/off，如果最后一个字段缺失会显示如下错误信息。

    ```
    missing data for column "tt"
    ```

  缺省值：false/off

须知

目前COPY指定此Option实际不会生效，即不会有相应的容错处理效果（不生效）。需要额外注意的是，打开此选项会导致解析器在CN数据解析阶段（即COPY错误表容错的涵盖范围）忽略此数据问题，而到DN重新报错，从而使得COPY错误表（打开LOG ERRORS REJECT LIMIT）在此选项打开的情况下无法成功捕获这类少列的数据异常。因此请不要指定此选项。

- DATE_FORMAT 'date_format_string'

  导入对于DATE类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法DATE格式。可参考**时间、日期处理函数和操作符**

  说明

  对于指定为ORACLE兼容类型的数据库，则DATE类型内建为TIMESTAMP类型。在导入的时候，若需指定格式，可以参考下面的timestamp_format参数。

- TIME_FORMAT 'time_format_string'

  导入对于TIME类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法TIME格式，不支持时区。可参考**时间、日期处理函数和操作符**。

- TIMESTAMP_FORMAT 'timestamp_format_string'

  导入对于TIMESTAMP类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法TIMESTAMP格式，不支持时区。可参考**时间、日期处理函数和操作符**。

- SMALLDATETIME_FORMAT 'smalldatetime_format_string'

  导入对于SMALLDATETIME类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

  取值范围：合法SMALLDATETIME格式。可参考**时间、日期处理函数和操作符**。

COPY FROM能够识别的特殊反斜杠序列如下所示。

- **\b**：反斜杠（ASCII 8）
- **\f**：换页（ASCII 12）
- **\n**：换行符（ASCII 10）
- **\r**：回车符（ASCII 13）
- **\t**：水平制表符（ASCII 9）
- **\v**：垂直制表符（ASCII 11）
- **\digits**：反斜杠后面跟着一到三个八进制数，表示ASCII值为该数的字符。
- **\xdigits**：反斜杠x后面跟着一个或两个十六进制位声明指定数值编码的字符。

示例

将ship_mode中的数据复制到/home/omm/ds_ship_mode.dat文件中：

```
COPY ship_mode TO '/home/omm/ds_ship_mode.dat';
```

将ship_mode 输出到stdout：

```
COPY ship_mode TO stdout;
```

创建ship_mode_t1表：

```
CREATE TABLE ship_mode_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
DISTRIBUTE BY HASH(SM_SHIP_MODE_SK );
```

从stdin复制数据到表ship_mode_t1：

```
COPY ship_mode_t1 FROM stdin;
```

从/home/omm/ds_ship_mode.dat文件复制数据到表ship_mode_t1：

```
COPY ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat';
```

从/home/omm/ds_ship_mode.dat文件复制数据到表ship_mode_t1，使用参数如下：导入格式为TEXT（format 'text'），分隔符为'\t'（delimiter E'\t'），忽略多余列（ignore_extra_data 'true'），不指定转义（noescaping 'true'）：

```
COPY ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' WITH(format 'text', delimiter E'\t',
ignore_extra_data 'true', noescaping 'true');
```

从/home/omm/ds_ship_mode.dat文件复制数据到表ship_mode_t1，使用参数如下：导入格式为FIXED（FIXED），指定定长格式（FORMATTER(SM_SHIP_MODE_SK(0, 2), SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10), SM_CARRIER(61,20), SM_CONTRACT(82,20)）），忽略多余列（ignore_extra_data），有数据头（header）：

```
COPY ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' FIXED FORMATTER(SM_SHIP_MODE_SK(0, 2),
SM_SHIP_MODE_ID(2,16), SM_TYPE(18,30), SM_CODE(50,10), SM_CARRIER(61,20), SM_CONTRACT(82,20))
header ignore_extra_data;
```

将ship_mode_t1导出为OBS的 '/bucket/path/'路径上的TEXT格式文件ds_ship_mode.dat。需要指定包含OBS访问信息的“server”option参数：

```
COPY ship_mode_t1 TO '/bucket/path/ds_ship_mode.dat' WITH (format 'text', encoding 'utf8', server
'obs_server');
```

将ship_mode_t1导出为OBS的 '/bucket/path/'路径上的的CSV格式文件。需要指定包含OBS访问信息的“server”option参数。其中文件包含标题行，包含BOM头，单文件最大行数1000行（超出1000行生成新的文件），自定义文件名前缀为“justprefix”：

```
COPY (select * from ship_mode_t1 where SM_SHIP_MODE_SK=1060) TO '/bucket/path/' WITH (format 'csv',
header 'on', encoding 'utf8', server 'obs_server', bom 'on', maxrow '1000', fileprefix 'justprefix');
```

删除ship_mode_t1：

```
DROP TABLE ship_mode_t1;
```


13.4 DELETE

功能描述

从指定的表里删除满足WHERE子句的行。如果WHERE子句不存在，将删除表中所有行，结果只保留表结构。

注意事项

- 要删除表中的数据，用户必须对它有DELETE权限。同样也必须有USING子句引用的表以及condition上读取的表的SELECT权限。
- 对于复制表，仅支持两种场景下的DELETE操作：
  - 有主键约束的场景。
  - 执行计划能下推的场景。
- 对于列存表，暂时不支持RETURNING子句。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING { * | { output_expr [ [ AS ] output_name ] } [, ...] } ];
```

参数说明

- **WITH [RECURSIVE] with_query [, ...]**

  用于声明一个或多个可以在主查询中通过名字引用的子查询，相当于临时表。

  如果声明了RECURSIVE，那么允许SELECT子查询通过名字引用它自己。

  其中with_query的详细格式为：

  ```
  with_query_name [ ( column_name [, ...] ) ] AS
  ( {select | values | insert | update | delete} )
  ```

  – with_query_name指定子查询生成的结果集名字，在查询中可使用该名称访问子查询的结果集。

  – column_name指定子查询结果集中显示的列名。

  – 每个子查询可以是SELECT，VALUES，INSERT，UPDATE或DELETE语句。

- **ONLY**

  如果指定ONLY则只有该表被删除；如果没有声明，则该表和它的所有子表将都被删除。

- **table_name**

  目标表的名字（可以有模式修饰）。

  取值范围：已存在的表名。

- **alias**

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。

- **using_list**

using子句。

- **condition**

  一个返回boolean值的表达式，用于判断哪些行需要被删除。

- **WHERE CURRENT OF cursor_name**

  当前不支持，仅保留语法接口。

- **output_expr**

  DELETE命令删除行之后计算输出结果的表达式。该表达式可以使用表的任意字段。可以使用*返回被删除行的所有字段。

- **output_name**

  一个字段的输出名称。

  取值范围：字符串，符合标识符命名规范。

示例

创建range分区表customer_address_bak：

```
DROP TABLE IF EXISTS customer_address_bak;
CREATE TABLE customer_address_bak
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
```

删除customer_address_bak中ca_address_sk小于14888的数据员：

```
DELETE FROM customer_address_bak WHERE ca_address_sk < 14888;
```

删除customer_address_bak中ca_address_sk为14891，14893和14895的数据：

```
DELETE FROM customer_address_bak WHERE ca_address_sk in (14891,14893,14895);
```

删除customer_address_bak中所有数据：

```
DELETE FROM customer_address_bak;
```

13.5 EXPLAIN

功能描述

显示SQL语句的执行计划。

执行计划将显示SQL语句所引用的表采用的扫描方式，如：简单的顺序扫描、索引扫描等。如果引用了多个表，执行计划还会显示使用的JOIN算法。

执行计划的最关键部分是语句的预计执行开销，这是计划生成器估算执行该语句将花费多长的时间。

若指定了ANALYZE选项，则该语句会被执行，然后根据实际的运行结果显示统计数据，包括每个计划节点内时间总开销（毫秒为单位）和实际返回的总行数。这对于判断计划生成器是否接近现实非常有用。

注意事项

在指定ANALYZE选项时，语句会被执行。如果用户使用EXPLAIN分析INSERT，UPDATE，DELETE，CREATE TABLE AS或EXECUTE语句，但不改动数据（执行这些语句会影响数据），可使用以下所示的方法：

```
START TRANSACTION;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

语法格式

- 显示SQL语句的执行计划，支持多种选项，对选项顺序无要求：
  ```
  EXPLAIN [ ( option [, ...] ) ] statement;
  ```
  其中选项option子句的语法为：
  ```
  ANALYZE [ boolean ] |
    ANALYSE [ boolean ] |
    VERBOSE [ boolean ] |
    COSTS [ boolean ] |
    CPU [ boolean ] |
    DETAIL [ boolean ] |
    NODES [ boolean ] |
    NUM_NODES [ boolean ] |
    BUFFERS [ boolean ] |
    TIMING [ boolean ] |
    PLAN [ boolean ] |
    FORMAT { TEXT | XML | JSON | YAML }
  ```
- 显示SQL语句的执行计划，且要按顺序给出选项：
  ```
  EXPLAIN { [ { ANALYZE | ANALYSE } ] [ VERBOSE ] | PERFORMANCE } statement;
  ```
- 显示复现SQL语句的执行计划所需的信息，通常用于定位问题。STATS选项必须单独使用：
  ```
  EXPLAIN ( STATS [ boolean ] ) statement;
  ```

参数说明

- **statement**

  指定要分析的SQL语句。
- **ANALYZE boolean | ANALYSE boolean**

  显示实际运行时间和其他统计数据。

  取值范围：
  - TRUE（缺省值）：显示实际运行时间和其他统计数据。
  - FALSE：不显示。
- **VERBOSE boolean**

  显示有关计划的额外信息。

  取值范围：
  - TRUE（缺省值）：显示额外信息。
  - FALSE：不显示。
- **COSTS boolean**

  包括每个规划节点的估计总成本，以及估计的行数和每行的宽度。

取值范围：

- TRUE（缺省值）：显示估计总成本和宽度。
- FALSE：不显示。

- **CPU boolean**

  打印CPU的使用情况的信息。

  取值范围：

  - TRUE（缺省值）：显示CPU的使用情况。
  - FALSE：不显示。

- **DETAIL boolean**

  打印DN上的信息。

  取值范围：

  - TRUE（缺省值）：打印DN的信息。
  - FALSE：不打印。

- **NODES boolean**

  打印query执行的节点信息。

  取值范围：

  - TRUE（缺省值）：打印执行的节点的信息。
  - FALSE：不打印。

- **NUM_NODES boolean**

  打印执行中的节点的个数信息。

  取值范围：

  - TRUE（缺省值）：打印DN个数的信息。
  - FALSE：不打印。

- **BUFFERS boolean**

  包括缓冲区的使用情况的信息。

  取值范围：

  - TRUE：显示缓冲区的使用情况。
  - FALSE（缺省值）：不显示。

- **TIMING boolean**

  包括实际的启动时间和花费在输出节点上的时间信息。

  取值范围：

  - TRUE（缺省值）：显示启动时间和花费在输出节点上的时间信息。
  - FALSE：不显示。

- **PLAN**

  是否将执行计划存储在plan_table中。当该选项开启时，会将执行计划存储在PLAN_TABLE中，不打印到当前屏幕，因此该选项为on时，不能与其他选项同时使用。

  取值范围：

  - ON（缺省值）：将执行计划存储在plan_table中，不打印到当前屏幕。执行成功返回EXPLAIN SUCCESS。
  - OFF：不存储执行计划，将执行计划打印到当前屏幕。

- **FORMAT**
  指定输出格式。
  取值范围：TEXT，XML，JSON和YAML。
  默认值：TEXT
- **PERFORMANCE**
  使用此选项时，即打印执行中的所有相关信息。
- **STATS boolean**
  打印复现SQL语句的执行计划所需的信息，包括对象定义、统计信息、配置参数等，通常用于定位问题。
  取值范围：
  - TRUE（缺省值）：显示复现SQL语句的执行计划所需的信息。
  - FALSE：不显示。

示例

创建一个表customer_address_p1：

```
CREATE TABLE customer_address_p1 AS TABLE customer_address;
```

修改explain_perf_mode为normal：

```
SET explain_perf_mode=normal;
```

显示表简单查询的执行计划：

```
EXPLAIN SELECT * FROM customer_address_p1;
```

```
postgres=> EXPLAIN SELECT * FROM customer_address_p1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes
(2 rows)
```

以JSON格式输出的执行计划（explain_perf_mode为normal时）：

```
EXPLAIN(FORMAT JSON) SELECT * FROM customer_address_p1;
```

```
postgres=> EXPLAIN(FORMAT JSON) SELECT * FROM customer_address_p1;
                    QUERY PLAN
--------------------------------------------------
 [                                               +
   {                                             +
     "Plan": {                                   +
       "Node Type": "Data Node Scan",            +
       "RemoteQuery name": "__REMOTE_FQS_QUERY__",+
       "Alias": "__REMOTE_FQS_QUERY__",          +
       "Startup Cost": 0.00,                     +
       "Total Cost": 0.00,                       +
       "Plan Rows": 0,                           +
       "Plan Width": 0,                          +
       "Nodes": "All datanodes"                  +
     }                                           +
   }                                             +
 ]
(1 row)
```

如果有一个索引，当使用一个带索引WHERE条件的查询，可能会显示一个不同的计划：


```
EXPLAIN SELECT * FROM customer_address_p1 WHERE ca_address_sk=10000;
```

```
postgres=> EXPLAIN SELECT * FROM customer_address_p1 WHERE ca_address_sk=10000;
                              QUERY PLAN
------------------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0 width=0)
   Node/s: All datanodes
(2 rows)
```

以YAML格式输出的执行计划（explain_perf_mode为normal时）：

```
EXPLAIN(FORMAT YAML) SELECT * FROM customer_address_p1 WHERE ca_address_sk=10000;
```

禁止开销估计的执行计划：

```
EXPLAIN(COSTS FALSE)SELECT * FROM customer_address_p1 WHERE ca_address_sk=10000;
```

带有聚集函数查询的执行计划：

```
EXPLAIN SELECT SUM(ca_address_sk) FROM customer_address_p1 WHERE ca_address_sk<10000;
```

删除表customer_address_p1：

```
DROP TABLE customer_address_p1;
```

相关链接

ANALYZE | ANALYSE

13.6 EXPLAIN PLAN

功能描述

通过EXPLAIN PLAN命令可以将查询执行的计划信息存储于PLAN_TABLE表中。与EXPLAIN命令不同的是，EXPLAIN PLAN仅将计划信息进行存储，而不会打印到屏幕。

语法格式

```
EXPLAIN PLAN
[ SET STATEMENT_ID = string ]
FOR statement ;
```

参数说明

- **PLAN**

  表示需要将计划信息存储于PLAN_TABLE中，存储成功将返回“EXPLAIN SUCCESS”。

- **STATEMENT_ID**

  用户可以对查询设置标签，输入的标签信息也将存储于PLAN_TABLE中。

  说明

  用户在执行EXPLAIN PLAN时，如果没有进行SET STATEMENT_ID，则默认为空值。同时，用户可输入的STATEMENT_ID最大长度为30个字节，超过长度将会产生报错。

注意事项

- EXPLAIN PLAN不支持在DN上执行。

- 对于执行错误的SQL无法进行计划信息的收集。
- PLAN_TABLE中的数据是session级生命周期并且session隔离和用户隔离，用户只能看到当前session、当前用户的数据。
- PLAN_TABLE无法与GDS外表进行关联查询。
- 对于不能下推的查询，无法收集到具体的object信息，object只能收集到REMOTE_QUERY或CTE等信息。详见**示例 2**。

示例 1

使用EXPLAIN PLAN收集SQL语句的执行计划，通常包括以下步骤：

步骤1 导入TPC-H样例数据。具体步骤参见**导入样例数据**。

步骤2 执行EXPLAN PLAN。

说明

执行EXPLAIN PLAN后会将计划信息自动存储于PLAN_TABLE中，不支持对PLAN_TABLE进行INSERT、UPDATE、ANALYZE等操作。

PLAN_TABLE详细介绍见视图PLAN_TABLE。

```
explain plan set statement_id='TPCH-Q4' for
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= '1993-07-01'::date
and o_orderdate < '1993-07-01'::date + interval '3 month'
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
```

步骤3 查询PLAN_TABLE。

```
SELECT * FROM PLAN_TABLE;
```

```
 statement_id | plan_id  | id |      operation      |   options   | object_name | object_type | object_owner |                projection
--------------+----------+----+---------------------+-------------+-------------+-------------+--------------+-------------------------------------------
 TPCH-Q4      | 16781167 |  1 | ROW ADAPTER         |             |             |             |              | ORDERS.O_ORDERPRIORITY, (PG_CATALOG.COUNT(*))
 TPCH-Q4      | 16781167 |  2 | VECTOR SORT         |             |             |             |              | ORDERS.O_ORDERPRIORITY, (PG_CATALOG.COUNT(*))
 TPCH-Q4      | 16781167 |  3 | VECTOR AGGREGATE    | HASHED      |             |             |              | ORDERS.O_ORDERPRIORITY, PG_CATALOG.COUNT(*)
 TPCH-Q4      | 16781167 |  4 | VECTOR STREAMING    | GATHER      |             |             |              | ORDERS.O_ORDERPRIORITY, (COUNT(*))
 TPCH-Q4      | 16781167 |  5 | VECTOR AGGREGATE    | HASHED      |             |             |              | ORDERS.O_ORDERPRIORITY, COUNT(*)
 TPCH-Q4      | 16781167 |  6 | VECTOR NESTED LOOPS | SEMI        |             |             |              | ORDERS.O_ORDERPRIORITY
 TPCH-Q4      | 16781167 |  7 | TABLE ACCESS        | CSTORE SCAN | ORDERS      | TABLE       | TPCH         | ORDERS.O_ORDERPRIORITY, ORDERS.O_ORDERKEY
 TPCH-Q4      | 16781167 |  8 | VECTOR MATERIALIZE  |             |             |             |              | LINEITEM.L_ORDERKEY
 TPCH-Q4      | 16781167 |  9 | TABLE ACCESS        | CSTORE SCAN | LINEITEM    | TABLE       | TPCH         | LINEITEM.L_ORDERKEY
```

步骤4 清理PLAN_TABLE表中的数据。

```
DELETE FROM PLAN_TABLE WHERE xxx;
```

----结束

示例 2

对于不能下推的查询，执行explain plan后plan_table中object仅收集到REMOTE_QUERY或CTE等信息。

优化器生成下发语句的计划，此时仅能收集到REMOTE_QUERY。

```
explain plan set statement_id = 'test remote query' for
select
current_user
from
customer;
```

查询PLAN_TABLE。

```
SELECT * FROM PLAN_TABLE;
```

```
   statement_id    | plan_id  | id |   operation    |  options  |   object_name    | object_type  | object_owner |  projection
-------------------+----------+----+----------------+-----------+------------------+--------------+--------------+--------------
 test remote query | 29360133 |  1 | NESTED LOOPS   | CARTESIAN |                  |              |              | 'apple'::name
 test remote query | 29360133 |  2 | DATA NODE SCAN |           | customer         | REMOTE_QUERY |              |
 test remote query | 29360133 |  3 | DATA NODE SCAN |           | customer_address | REMOTE_QUERY |              |
(3 rows)
```

13.7 LOCK

功能描述

LOCK TABLE获取表级锁。

当自动获取引用表的命令的锁时，GaussDB(DWS)会始终使用限制最小的锁模式。如果用户需要一种更为严格的锁模式，可以使用LOCK命令。例如，某个应用是在Read Committed隔离级别上运行事务，并且需要保证表中的数据在事务运行期间保持稳定。为实现这个目的，则可以在查询之前对表使用SHARE锁模式进行锁定。这样将防止并发数据更改，并确保后续的查询可以读到已提交的持久化的数据。因为SHARE锁模式与任何写操作需要的ROW EXCLUSIVE模式冲突，并且LOCK TABLE name IN SHARE MODE语句将等到所有当前持有ROW EXCLUSIVE模式锁的事务提交或回滚后才能执行。因此，一旦获得该锁，就不会存在未提交的写操作，此外其他操作也只能等到该锁释放之后才能开始。

注意事项

- LOCK TABLE只能在一个事务块的内部有用，因为锁在事务结束时就会被释放。出现在任意事务块外面的LOCK TABLE都会报错。
- 如果没有声明锁模式，缺省为最严格的模式ACCESS EXCLUSIVE。
- LOCK TABLE ... IN ACCESS SHARE MODE需要在目标表上有SELECT权限。所有其他形式的LOCK需要UPDATE和/或DELETE权限。
- 没有UNLOCK TABLE命令，锁总是在事务结束时释放。
- LOCK TABLE只处理表级的锁，因此那些带“ROW”字样的锁模式都是有歧义的。这些模式名字通常可理解为用户试图在一个被锁定的表中获取行级的锁。同样，ROW EXCLUSIVE模式也是一个可共享的表级锁。注意，只要是涉及到LOCK TABLE ，所有锁模式都有相同的语意，区别仅在于规则中锁与锁之间是否冲突，规则请参见**表13-1**。

语法格式

```
LOCK [ TABLE ] {[ ONLY ] name [, ...]| {name [ * ]} [, ...]}
    [ IN {ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE} MODE ]
    [ NOWAIT ];
```

参数说明

表 13-1 冲突的锁模式

请求的锁模式/当前锁模式	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE	-	-	-	-	-	-	-	X
ROW SHARE	-	-	-	-	-	-	X	X
ROW EXCLUSIVE	-	-	-	-	X	X	X	X
SHARE UPDATE EXCLUSIVE	-	-	-	X	X	X	X	X
SHARE	-	-	X	X	-	X	X	X
SHARE ROW EXCLUSIVE	-	-	X	X	X	X	X	X
EXCLUSIVE	-	X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

LOCK的参数说明如下所示：

- **name**

  要锁定的表的名字，可以有模式修饰。

  LOCK TABLE命令中声明的表的顺序就是上锁的顺序。

  取值范围：已存在的表名。

- **ONLY**

  如果指定ONLY只有该表被锁定，如果没有声明该表和他的所有子表将都被锁定。

- **ACCESS SHARE**

  ACCESS锁只允许对表进行读取，而禁止对表进行修改。所有对表进行读取而不修改的SQL语句都会自动请求这种锁。例如，SELECT命令会自动在被引用的表上请求一个这种锁。

- **ROW SHARE**

  ROW SHARE锁允许对表进行并发读取，禁止对表进行其他操作。

  SELECT FOR UPDATE和SELECT FOR SHARE命令会自动在目标表上请求ROW SHARE锁（且所有被引用但不是FOR SHARE/FOR UPDATE的其他表上，还会自动加上ACCESS SHARE锁）。

- **ROW EXCLUSIVE**

  与ROW SHARE锁不同，ROW EXCLUSIVE允许并发读取表，也允许修改表中的数据。UPDATE，DELETE，INSERT命令会自动在目标表上请求这个锁（且所有被引用的其他表上还会自动加上的ACCESS SHARE锁）。通常情况下，所有会修改表数据的命令都会请求表的ROW EXCLUSIVE锁。

- **SHARE UPDATE EXCLUSIVE**

  这个模式保护一个表的模式不被并发修改，以及禁止在目标表上执行垃圾回收命令（VACUUM ）。

  VACUUM（不带FULL选项），ANALYZE，CREATE INDEX CONCURRENTLY命令会自动请求这样的锁。

- **SHARE**

  SHARE锁允许并发的查询，但是禁止对表进行修改。

  CREATE INDEX（不带CONCURRENTLY选项）语句会自动请求这种锁。

- **SHARE ROW EXCLUSIVE**

  SHARE ROW EXCLUSIVE锁禁止对表进行任何的并发修改，而且是独占锁，因此一个会话中只能获取一次。

  任何SQL语句都不会自动请求这个锁模式。

- **EXCLUSIVE**

  EXCLUSIVE锁允许对目标表进行并发查询，但是禁止任何其他操作。

  这个模式只允许并发加ACCESS SHARE锁，也就是说，只有对表的读动作可以和持有这个锁模式的事务并发执行。

  任何SQL语句都不会在用户表上自动请求这个锁模式。然而在某些操作的时候，会在某些系统表上请求它。

- **ACCESS EXCLUSIVE**

  这个模式保证其所有者（事务）是可以访问该表的唯一事务。

  ALTER TABLE，DROP TABLE，TRUNCATE，REINDEX，CLUSTER，VACUUM FULL命令会自动请求这种锁。

  在LOCK TABLE命令没有明确声明需要的锁模式时，它是缺省锁模式。

- **NOWAIT**

  声明LOCK TABLE不去等待任何冲突的锁释放，如果无法立即获取该锁，该命令退出并且发出一个错误信息。

  在不指定NOWAIT的情况下获取表级锁时，如果有其他互斥锁存在的话，则等待其他锁的释放。

示例

向一个外键表上插入数据时，在有主键的表上使用SHARE锁：

```
DROP TABLE IF EXISTS customer_address;
CREATE TABLE customer_address
(
    ca_address_sk       INTEGER                 NOT NULL   ,
    ca_address_id       CHARACTER(16)           NOT NULL   ,
    ca_street_number    CHARACTER(10)                      ,
    ca_street_name      CHARACTER varying(60)              ,
    ca_street_type      CHARACTER(15)                      ,
    ca_suite_number     CHARACTER(10)
)
DISTRIBUTE BY HASH (ca_address_sk)
PARTITION BY RANGE(ca_address_sk)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
START TRANSACTION;

LOCK TABLE customer_address IN SHARE MODE;

SELECT ca_address_sk FROM customer_address WHERE ca_address_sk=5;

COMMIT;
```

在执行删除操作时对一个有主键的表进行 SHARE ROW EXCLUSIVE锁：

```
DROP TABLE IF EXISTS customer;
CREATE TABLE customer AS TABLE customer_address;

START TRANSACTION;

LOCK TABLE customer IN SHARE ROW EXCLUSIVE MODE;

DELETE FROM customer WHERE ca_address_sk IN(SELECT ca_address_sk FROM customer WHERE
ca_address_sk < 6 );

DELETE FROM customer WHERE ca_address_sk = 7;

COMMIT;
```

13.8 MERGE INTO

功能描述

通过MERGE INTO语句，将目标表和源表中数据针对关联条件进行匹配，若关联条件匹配时对目标表进行UPDATE，无法匹配时对目标表执行INSERT。此语法可以很方便地用来合并执行UPDATE和INSERT，避免多次执行。

注意事项

- 进行MERGE INTO操作的用户需要同时拥有目标表的UPDATE和INSERT权限，以及源表的SELECT权限。
- 不支持PREPARE。
- 不支持重分布过程中MERGE INTO。
- 不支持对包含触发器的目标表执行MERGE INTO。

- 对roundrobin表执行MERGE INTO时，推荐关闭GUC参数allow_concurrent_tuple_update，否则会不支持部分MERGE INTO语句。

语法格式

```
MERGE INTO table_name [ [ AS ] alias ]
USING { { table_name | view_name } | subquery } [ [ AS ] alias ]
ON ( condition )
[
  WHEN MATCHED THEN
  UPDATE SET { column_name = { expression | DEFAULT } |
         ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
  [ WHERE condition ]
]
[
  WHEN NOT MATCHED THEN
  INSERT { DEFAULT VALUES |
  [ ( column_name [, ...] ) ] VALUES ( { expression | DEFAULT } [, ...] ) [, ...] [ WHERE condition ] }
];
```

参数说明

- **INTO子句**

  指定正在更新或插入的目标表。

  - **table_name**

    目标表的表名。

  - **alias**

    目标表的别名。

    取值范围：字符串，符合标识符命名规范。

- **USING子句**

  指定源表，源表可以为表、视图或子查询。

- **ON子句**

  关联条件，用于指定目标表和源表的关联条件。不支持更新关联条件中的字段。

- **WHEN MATCHED子句**

  当源表和目标表中数据针对关联条件可以匹配上时，选择WHEN MATCHED子句进行UPDATE操作。

  不支持更新分布列。不支持更新系统表、系统列。

- **WHEN NOT MATCHED子句**

  当源表和目标表中数据针对关联条件无法匹配时，选择WHEN NOT MATCHED子句进行INSERT操作。

  不支持INSERT子句中包含多个VALUES。

  WHEN MATCHED和WHEN NOT MATCHED子句顺序可以交换，可以缺省其中一个，但不能同时缺省，不支持同时指定两个WHEN MATCHED或WHEN NOT MATCHED子句。

- **DEFAULT**

  用对应字段的缺省值填充该字段。

  如果没有缺省值，则为NULL。

- **WHERE condition**

  UPDATE子句和INSERT子句的条件，只有在条件满足时才进行更新操作，可缺省。不支持WHERE条件中引用系统列。

示例

创建目标表products和源表newproducts，并插入数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products
(
product_id INTEGER,
product_name VARCHAR2(60),
category VARCHAR2(60)
);

INSERT INTO products VALUES (1501, 'vivitar 35mm', 'electrncs');
INSERT INTO products VALUES (1502, 'olympus is50', 'electrncs');
INSERT INTO products VALUES (1600, 'play gym', 'toys');
INSERT INTO products VALUES (1601, 'lamaze', 'toys');
INSERT INTO products VALUES (1666, 'harry potter', 'dvd');

DROP TABLE IF EXISTS newproducts;
CREATE TABLE newproducts
(
product_id INTEGER,
product_name VARCHAR2(60),
category VARCHAR2(60)
);

INSERT INTO newproducts VALUES (1502, 'olympus camera', 'electrncs');
INSERT INTO newproducts VALUES (1601, 'lamaze', 'toys');
INSERT INTO newproducts VALUES (1666, 'harry potter', 'toys');
INSERT INTO newproducts VALUES (1700, 'wait interface', 'books');
```

进行MERGE INTO操作：

```
MERGE INTO products p
USING newproducts np
ON (p.product_id = np.product_id)
WHEN MATCHED THEN
  UPDATE SET p.product_name = np.product_name, p.category = np.category WHERE p.product_name !=
'play gym'
WHEN NOT MATCHED THEN
  INSERT VALUES (np.product_id, np.product_name, np.category) WHERE np.category = 'books';
```

查询更新后的结果：

```
SELECT * FROM products ORDER BY product_id;
```

```
postgres=> SELECT * FROM products ORDER BY product_id;
 product_id |  product_name  | category
------------+----------------+-----------
       1501 | vivitar 35mm   | electrncs
       1502 | olympus camera | electrncs
       1600 | play gym       | toys
       1601 | lamaze         | toys
       1666 | harry potter   | toys
       1700 | wait interface | books
(6 rows)
```

删除表：

```
DROP TABLE products;
DROP TABLE newproducts;
```

13.9 INSERT 和 UPSERT

13.9.1 INSERT

功能描述

向表中添加一行或多行数据。

注意事项

- 只有拥有表INSERT权限的用户，才可以向表中插入数据。
- 如果使用RETURNING子句，用户必须要有该表的SELECT权限。
- 如果使用QUERY子句插入来自查询里的数据行，用户还需要拥有在查询里使用的表的SELECT权限。
- 如果使用OVERWRITE子句覆盖式插入数据，用户还需要拥有该表的SELECT和TRUNCATE权限。
- 当连接到TD兼容的数据库时，td_compatible_truncation参数设置为on时，将启用超长字符串自动截断功能，在后续的INSERT语句中（不包含外表的场景下），对目标表中char和varchar类型的列上插入超长字符串时，系统会自动按照目标表中相应列定义的最大长度对超长字符串进行截断。

  说明

  如果向字符集为字节类型编码（SQL_ASCII，LATIN1等）的数据库中插入多字节字符数据（如汉字等），且字符数据跨越截断位置，这种情况下，按照字节长度自动截断，自动截断后会在尾部产生非预期结果。如果用户有对于截断结果正确性的要求，建议用户采用UTF8等能够按照字符截断的输入字符集作为数据库的编码集。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT [ IGNORE | OVERWRITE ] INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
   { DEFAULT VALUES
   | VALUES {( { expression | DEFAULT } [, ...] ) }[, ...]
   | query }
   [ ON DUPLICATE KEY duplicate_action | ON CONFLICT [ conflict_target ] conflict_action ]
   [ RETURNING {* | {output_expression [ [ AS ] output_name ] }[, ...]} ];

where duplicate_action can be:

   UPDATE { column_name = { expression | DEFAULT } |
            ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] )
          } [, ...]

and conflict_target can be one of:

   ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] ) [ WHERE
index_predicate ]
   ON CONSTRAINT constraint_name

and conflict_action is one of:

   DO NOTHING
   DO UPDATE SET { column_name = { expression | DEFAULT } |
                   ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] )
                 } [, ...]
             [ WHERE condition ]
```

参数说明

- **WITH [RECURSIVE] with_query [, ...]**

用于声明一个或多个可以在主查询中通过名字引用的子查询，相当于临时表。

如果声明了RECURSIVE，那么允许SELECT子查询通过名字引用它自己。

其中with_query的详细格式为：

```
with_query_name [ ( column_name [, ...] ) ] AS
( {select | values | insert | update | delete} )
```

– with_query_name指定子查询生成的结果集名字，在查询中可使用该名称访问子查询的结果集。

– column_name指定子查询结果集中显示的列名。

– 每个子查询可以是SELECT，VALUES，INSERT，UPDATE或DELETE语句。

- IGNORE

  用于主键或者唯一约束冲突时忽略冲突的数据。

  详细介绍参见**UPSERT**。

- OVERWRITE

  用于标识覆盖式插入方式，使用此种插入方式执行结束后，目标原数据被清空，只存在新插入的数据。

  OVERWRITE支持指定列插入的功能，其他列为默认值，若无默认值则为NULL。

  须知

  - OVERWRITE不要和INSERT INTO这类实时写入的操作并发，否则实时写入数据有被意外清理的风险。
  - OVERWRITE适用于大批量数据导入场景，不建议用于少量数据的插入场景。
  - 避免对同一张表执行并发insert overwrite操作，否则会出现类似报错“tuple concurrently updated.”。
  - 如果集群正在扩缩容，且INSERT OVERWRITE的写入表需要执行数据重分布，则INSERT OVERWRITE会清除当前数据，并自动将插入的数据按扩缩容后的节点来进行数据分布。如果INSERT OVERWRITE和该表的数据重分布过程同时执行，INSERT OVERWRITE会中断该表的数据重分布过程。

- **table_name**

  要插入数据的目标表名。

  取值范围：已存在的表名。

- **AS**

  用于给目标表table_name指定别名。alias即为别名的名字。

- **column_name**

  目标表中的字段名：

  – 字段名可以有子字段名或者数组下标修饰。

  – **没有在字段列表中出现的每个字段，将由系统默认值，或者声明时的默认值填充，若都没有则用NULL填充。例如，向一个复合类型中的某些字段插入数据的话，其他字段将是NULL。**

  – 目标字段（column_name）可以按顺序排列。如果没有列出任何字段，则默认全部字段，且顺序为表声明时的顺序。

  – 如果value子句和query中只提供了N个字段，则目标字段为前N个字段。

- value子句和query提供的值在表中从左到右关联到对应列。

取值范围：已存在的字段名。

- **expression**

  赋予对应column的一个有效表达式或值：

  - 向表中字段插入单引号 时需要使用单引号自身进行转义。
  - 如果插入行的表达式不是正确的数据类型，系统试图进行类型转换，若转换不成功，则插入数据失败，系统返回错误信息。

  示例：

```
CREATE TABLE tt01 (id int,content varchar(50));
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using round-robin as the distribution mode by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE

INSERT INTO tt01 values (1,'Jack say ''hello''');
INSERT 0 1
INSERT INTO tt01 values (2,'Rose do 50%');
INSERT 0 1
INSERT INTO tt01 values (3,'Lilei say ''world''');
INSERT 0 1
INSERT INTO tt01 values (4,'Hanmei do 100%');
INSERT 0 1

SELECT * FROM tt01;
 id |      content
----+-------------------
  3 | Lilei say 'world'
  4 | Hanmei do 100%
  1 | Jack say 'hello'
  2 | Rose do 50%
(4 rows)
```

- **DEFAULT**

  对应字段名的缺省值。如果没有缺省值，则为NULL。

- **query**

  一个查询语句（SELECT语句），将查询结果作为插入的数据。

- **ON DUPLICATE KEY**

  用于主键或者唯一约束冲突时更新冲突的数据。

  duplicate_action指定更新列和更新的数据。

  详细介绍参见**UPSERT**。

- **ON CONFLICT**

  用于主键或者唯一约束冲突时忽略或者更新冲突的数据。

  conflict_target用于指定列名index_column_name 、包含多个列名的表达式index_expression 或者约束名字constraint_name。作用是用于从列名、包含多个列名的表达式或者约束名推断是否有唯一索引。其中index_column_name和index_expression遵循CREATE INDEX的索引列格式。

  conflict_action 指定主键或者唯一约束冲突时执行的策略。有两种：

  - DO NOTHING冲突忽略。
  - DO UPDATE SET冲突更新。 后面指定更新列和更新的数据。

  详细介绍参见**UPSERT**。

- **RETURNING**

  返回实际插入的行，RETURNING列表的语法与SELECT的输出列表一致。

- **output_expression**

  INSERT命令在每一行都被插入之后用于计算输出结果的表达式。

  取值范围：该表达式可以使用table的任意字段。可以使用*返回被插入行的所有字段。

- **output_name**

  字段的输出名称。

  取值范围：字符串，符合标识符命名规范。

示例

创建表reason_t1：

```
DROP TABLE IF EXISTS reason_t1;
CREATE TABLE reason_t1
(
   TABLE_SK          INTEGER               ,
   TABLE_ID          VARCHAR(20)           ,
   TABLE_NA          VARCHAR(20)
);
```

向表中插入一条记录：

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NA) VALUES (1, 'S01', 'StudentA');
```

向表中插入一条记录，和上一条语法等效：

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA');
```

向表中插入TABLE_SK小于1的记录：

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 WHERE TABLE_SK < 1;
```

向表中插入多条记录：

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
SELECT * FROM reason_t1 ORDER BY 1;
```

使用INSERT OVERWRITE更新表中的数据，即覆盖式插入数据：

```
INSERT OVERWRITE INTO reason_t1 values (4, 'S02', 'StudentB');
SELECT * FROM reason_t1 ORDER BY 1;
```

将表reason_t1的数据插入到表reason_t1中：

```
INSERT INTO reason_t1 SELECT * FROM reason_t1;
```

对独立的字段明确缺省值：

```
INSERT INTO reason_t1 VALUES (5, 'S03', DEFAULT);
```

将一个表中的部分数据插入到另一个表中：先通过WITH子查询得到一张临时表temp_t，然后将临时表temp_t中的所有数据插入另一张表reason_t1中：

```
WITH temp_t AS (SELECT * FROM reason_t1) INSERT INTO reason_t1 SELECT * FROM temp_t ORDER BY 1;
```

13.9.2 UPSERT

功能描述

向表中添加一行或多行数据。当出现主键或者唯一约束冲突时更新或者忽略冲突的数据。

须知

UPSERT语法仅8.1.1及以上版本支持。

注意事项

- 当在列存表上执行UPSERT时，建议开启DELTA表。开启DELTA表能够有效防止执行UPSERT时产生小CU（大量的小CU会导致空间膨胀和查询性能差）。
- 对于列存表上的UPSERT、UPDATE、DELETE并发场景，由于并发更新到同一个CU时需要等待CU锁，无法支持这几个操作的并发执行，开启DELTA也无法解决该问题。
- 只有拥有表INSERT、UPDATE权限的用户，才可以通过UPSERT语句向表中插入或更新数据。
- 目标表上必须包含主键或者唯一索引才可以执行UPSERT的冲突更新语句。
- 所有的唯一索引都不可用时不能执行UPSERT的冲突更新语句，重建索引后可以正常执行。
- 可能存在分布式死锁导致查询hang问题。

  说明

  例如场景：一个事务中或者通过JDBC(setAutoCommit(false))批量执行多条UPSERT语句，多个类似任务同时执行。

  可能产生结果：由于不同线程在不同节点更新顺序可能不同，在存在并发更新同一行的场景里可能会有死锁问题。

  解决办法：

  1. 减小GUC参数lockwait_timeout 值（默认20min）。分布式死锁会等待lockwait_timeout 时间然后报错。通过减小此参数的数值，降低死锁造成的业务等待时间。
  2. 保证主键相同的数据从只从一个数据库连接导入数据库。可以并发执行UPSERT语句。
  3. 每个事务中只执行一条UPSERT语句。可以并发执行UPSERT语句。
  4. 单线程执行多条UPSERT语句，不能并发执行UPSERT语句。

  如上解决办法中，方法1只能降低等待时间，无法解决死锁问题。在业务中有UPSERT语句时，仍建议减小此参数数值；方法2、3、4均可以解决死锁问题，但建议采用方法2，其性能优于其他两个方法。
- 不能更新分布列（**例外：当分布键与更新值相同时，分布列可更新**）。

```
CREATE TABLE t1(dist_key int PRIMARY KEY, a int, b int);
INSERT INTO t1 VALUES(1,2,3) ON CONFLICT(dist_key) DO UPDATE SET dist_key =
EXCLUDED.dist_key, a = EXCLUDED.a + 1;
INSERT INTO t1 VALUES(1,2,3) ON CONFLICT(dist_key) DO UPDATE SET dist_key = dist_key, a =
EXCLUDED.a + 1;
```

- 不支持对包含触发器（触发事件为INSERT或UPDATE）的目标表执行UPSERT语句。
- 不支持对可更新视图执行UPSERT语句。
- UPDATE子句、UPDATE中WHERE子句或者索引条件表达式不能包含不下推函数。
- 不支持延迟唯一索引。
- 通过INSERT INTO SELECT语句执行UPSERT的更新操作时，需要注意SELECT语句的查询结果顺序。在分布式环境中未使用ORDER BY语句时每次执行相同的SELECT语句返回结果顺序可能不一样，这会导致UPSERT语句的执行结果不符合预期。

- 不支持多次更新。插入多组数据间如果有冲突，则会报错（**例外：当查询计划是PGXC计划时**）。

```
CREATE TABLE t1(id int PRIMARY KEY, a int, b int);
-- STREAM计划
EXPLAIN (COSTS OFF) INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a =
EXCLUDED.a + 1;
                  QUERY PLAN
-------------------------------------------------
 id |               operation
----+-----------------------------------------
  1 | ->  Streaming (type: GATHER)
  2 |    ->  Insert on t1
  3 |       ->  Streaming(type: REDISTRIBUTE)
  4 |          ->  Values Scan on "*VALUES*"
Predicate Information (identified by plan id)
-----------------------------------------------
  2 --Insert on t1
        Conflict Resolution: UPDATE
        Conflict Arbiter Indexes: t1_pkey
  ====== Query Summary =====
------------------------------
System available mem: 819200KB
Query Max mem: 819200KB
Query estimated mem: 3104KB
(18 rows)
INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a = EXCLUDED.a + 1;
ERROR:  INSERT ON CONFLICT DO UPDATE command cannot affect row a second time
HINT:  Ensure that no rows proposed for insertion within the same command have duplicate
constrained values.
-- 关闭stream，生成PGXC计划
set enable_stream_operator = off;
EXPLAIN (COSTS OFF) INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a =
EXCLUDED.a + 1;
                QUERY PLAN
-----------------------------------------------
 id |           operation
----+-----------------------------------
  1 | ->  Insert on t1
  2 |    ->  Values Scan on "*VALUES*"
Predicate Information (identified by plan id)
-----------------------------------------------
  1 --Insert on t1
        Conflict Resolution: UPDATE
        Conflict Arbiter Indexes: t1_pkey
        Node expr: id
(11 rows)
INSERT INTO t1 VALUES(1,2,3),(1,5,6) ON CONFLICT(id) DO UPDATE SET a = EXCLUDED.a + 1;
INSERT 0 2
```

语法格式

详细介绍请参见INSERT的**语法格式**。有两种UPSERT语法格式：

表 13-2 UPSERT 语法格式

语法格式	冲突更新	冲突忽略
第一种：不指定索引	INSERT INTO ON DUPLICATE KEY UPDATE	INSERT IGNORE INSERT INTO ON CONFLICT DO NOTHING

语法格式	冲突更新	冲突忽略
第二种：从指定列名或者约束上可以推断唯一约束	INSERT INTO ON CONFLICT(...) DO UPDATE SET INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO UPDATE SET	INSERT INTO ON CONFLICT(...) DO NOTHING INSERT INTO ON CONFLICT ON CONSTRAINT con_name DO NOTHING

第一种不指定索引。会在所有主键或唯一索引上检查冲突，有冲突就会忽略或者更新。

第二种指定索引。会从ON CONFLICT子句中指定列名、包含列名的表达式或者约束名上推断主键或者唯一索引。

- 唯一索引推断

  对于第二种语法形式，通过指定列名或者约束名推断主键或者唯一索引。列名可以是单一列名，或者由多个列名组成的表达式，比如（column1，column2，column3）。

  由于创建索引时可以指定collation和opclass，所以此处列名后也可以指定。

  COLLATE collation指定列的排序规则。opclass指定操作符类的名字。具体参考**CREATE INDEX**。

  从指定列名的表达式中推断出唯一索引，整体原则是判断某唯一索引是否能够恰好完全包含conflict_target指定的列名。

  - 如果没有指定collation和opclass，那么只要列或者列名的表达式相同（不管索引列上指定的collation和opclass是什么），都认为匹配。
  - 如果指定collation和opclass，那么需要与索引的collation和opclass匹配才可以。

- UPDATE子句

UPDATE子句可以通过VALUES(colname)或者EXCLUDED.colname引用插入的数据。EXCLUDED表示因冲突原本该排除的数据行。示例如下：

```
CREATE TABLE t1(id int PRIMARY KEY, a int, b int);
INSERT INTO t1 VALUES(1,1,1);
-- 对于冲突行，把a列修改为目标表a列值加1，更新为(1,2,1)
INSERT INTO t1 VALUES(1,10,20) ON CONFLICT(id) DO UPDATE SET a = a + 1;
-- EXCLUDED.a 表示引用插入值的a列。本例中为10。
-- 对于冲突行，把a列修改为引用插入的a列值。更新为(1,11,1)
INSERT INTO t1 VALUES(1,10,20) ON CONFLICT(id) DO UPDATE SET a = EXCLUDED.a + 1;
```

- WHERE子句
  - 用于在数据冲突时，判断是否满足指定条件。如果满足，则更新冲突数据。否则忽略。
  - 只有第二种语法形式的冲突更新语法可以指定WHERE子句。即 INSERT INTO ON CONFLICT(...) DO UPDATE SET WHERE

语法使用注意事项：

- **表13-2**中几种语法形式不能在同一个语句中一起使用。
- 不支持与WITH子句同时使用。
- 不支持与INSERT OVERWRITE同时使用。

- UPDATE子句和UPDATE的WHERE子句不能有子查询。
- UPDATE子句中VALUES(colname)用法不支持外层嵌套函数，即不支持类似sqrt(VALUES(colname))用法。如需支持，使用EXCLUDED.colname语法。
- INSERT INTO ON CONFLICT(...) DO UPDATE必须有conflict_target。即必须指定列或者约束名。

示例

创建表reason_t2，并向表中插入数据：

```
DROP TABLE IF EXISTS reason_t2;
CREATE TABLE reason_t2
(
  a    int primary key,
  b    int,
  c    int
);

INSERT INTO reason_t2 VALUES (1, 2, 3);
SELECT * FROM reason_t2 ORDER BY 1;
```

向表reason_t2中插入两条数据，一条有冲突，一条无冲突。有冲突的数据进行忽略，无冲突的数据进行插入：

```
INSERT INTO reason_t2 VALUES (1, 4, 5),(2, 6, 7) ON CONFLICT(a) DO NOTHING;
SELECT * FROM reason_t2 ORDER BY 1;
```

向表reason_t2中插入数据，一条有冲突，一条无冲突。有冲突的数据进行更新，无冲突的数据进行插入：

```
INSERT INTO reason_t2 VALUES (1, 4, 5),(3, 8, 9) ON CONFLICT(a) DO UPDATE SET b = EXCLUDED.b, c =
EXCLUDED.c;
SELECT * FROM reason_t2 ORDER BY 1;
```

根据过滤条件筛选被更新的行：

```
INSERT INTO reason_t2 VALUES (2, 7, 8) ON CONFLICT (a) DO UPDATE SET b = excluded.b, c = excluded.c
WHERE reason_t2.c = 7;
SELECT * FROM reason_t2 ORDER BY 1;
```

向表reason_t中插入数据，有冲突的数据进行更新并调整更新映射关系，即c列更新到b，b列更新到c：

```
INSERT INTO reason_t2 VALUES (1, 2, 3) ON CONFLICT (a) DO UPDATE SET b = excluded.c, c = excluded.b;
SELECT * FROM reason_t2 ORDER BY 1;
```

13.10 UPDATE

功能描述

更新表中的数据。UPDATE修改满足条件的所有行中指定的字段值，WHERE子句声明条件，SET子句指定的字段会被修改，没有出现的字段则保持它们的原值。

注意事项

- 要修改表，用户必须对该表有UPDATE权限。
- 对expression或condition条件里涉及到的任何表要有SELECT权限。
- 不允许对表的分布列（distribute column）进行修改。

- 对于列存表，暂时不支持RETURNING子句。
- 列存表不支持结果不确定的更新（non-deterministic update）。试图对列存表用多行数据更新一行时会报错。
- 列存表的更新操作，旧记录空间不会回收，需要执行VACUUM FULL table_name进行清理。
- UPDATE操作频繁的表不建议创建为复制表。
- 对于列存表，支持轻量化UPDATE操作。轻量化UPDATE只重写更新列，减少空间使用量。列存轻量化UPDATE通过GUC参数enable_light_colupdate控制是否开启。
- 列存轻量化UPDATE在以下场景不能使用：更新索引列，更新主键列，更新分区列，更新PCK列和在线扩容，会自动转化为普通UPDATE方式。

语法格式

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
SET {column_name = { expression | DEFAULT }
    |( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] ) |sub_query }}[, ...]
    [ FROM from_list] [ WHERE condition ]
    [ RETURNING {*
                | {output_expression [ [ AS ] output_name ]} [, ...] }];

where sub_query can be:
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
```

参数说明

- **table_name**

  要更新的表名，可以使用模式修饰。

  取值范围：已存在的表名称。

- **alias**

  目标表的别名。

  取值范围：字符串，符合标识符命名规范。

- **column_name**

  要修改的字段名。

  支持使用目标表的表名加字段名来引用这个字段。例如：

  ```
  UPDATE foo SET foo.col_name = 'GaussDB';
  ```

  支持使用目标表的别名加字段名来引用这个字段。例如：

  ```
  UPDATE foo AS f SET f.col_name = 'GaussDB';
  ```

  取值范围：已存在的字段名。

- **expression**

  赋给字段的值或表达式。

- **DEFAULT**

  用对应字段的缺省值填充该字段。

  如果没有缺省值，则为NULL。

- **sub_query**

  子查询。

  使用同一数据库里其他表的信息来更新一个表可以使用子查询的方法。其中SELECT子句具体介绍请参考**SELECT**。

- **from_list**

  一个表的表达式列表，允许在WHERE条件里使用其他表的字段。与在一个SELECT语句的FROM子句里声明表列表类似。

  须知

  目标表绝对不能出现在from_list里，除非在使用一个自连接（此时它必须以from_list的别名出现）。

- **condition**

  一个返回boolean类型结果的表达式。只有这个表达式返回true的行才会被更新。

- **output_expression**

  在所有需要更新的行都被更新之后，UPDATE命令用于计算返回值的表达式。

  取值范围：使用任何table以及FROM中列出的表的字段。*表示返回所有字段。

- **output_name**

  字段的返回名称。

示例

创建表reason：

```
DROP TABLE IF EXISTS reason;
CREATE TABLE reason
(
  r_reason_sk    int,
  r_reason_desc    char(20),
  r_reason_id    char(20)
);

INSERT INTO reason VALUES (1, '2', '3');
```

直接更新所有记录的值：

```
UPDATE reason SET r_reason_sk = r_reason_sk * 2;
```

不含WHERE子句表示更新所有r_reason_sk的值：

```
UPDATE reason SET r_reason_sk = r_reason_sk + 100;
```

将表reason中r_reason_desc为reason2的r_reason_sk重新定义：

```
UPDATE reason SET r_reason_sk = 5 WHERE r_reason_desc = 'reason2';
```

将表reason中r_reason_sk为2的r_reason_sk重新定义：

```
UPDATE reason SET r_reason_sk = r_reason_sk + 100 WHERE r_reason_sk = 2;
```

将表reason中r_reason_sk大于2的课程编号全部重新定义：

```
UPDATE reason SET r_reason_sk = 201 WHERE r_reason_sk > 2;
```

可以在一个UPDATE命令中更新多个字段，方法是在SET子句中列出更多赋值，比如：

```
UPDATE reason SET r_reason_sk = 5, r_reason_desc = 'reason5' WHERE r_reason_id = 'fourth';
```

13.11 VALUES

功能描述

根据给定的值表达式计算一个或一组行的值。它通常用于在一个较大的命令内生成一个“常数表”。

注意事项

- 应当避免使用VALUES返回数量非常大的结果行，否则可能会遭遇内存耗尽或者性能低下。出现在INSERT中的VALUES是一个特殊情况，因为目标字段类型可以从INSERT的目标表获知，并不需要通过扫描VALUES列表来推测，所以在此情况下可以处理非常大的结果行。
- 如果指定了多行，那么每一行都必须拥有相同的元素个数。

语法格式

```
VALUES {( expression [, ...] )} [, ...]
    [ ORDER BY { sort_expression [ ASC | DESC | USING operator ] } [, ...] ]
    [ { [ LIMIT { count | ALL } ] [ OFFSET start [ ROW | ROWS ] ] } | { LIMIT start, { count | ALL } } ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ];
```

参数说明

- **expression**

  用于计算或插入结果表指定地点的常量或者表达式。

  在一个出现在INSERT顶层的VALUES列表中，expression可以被DEFAULT替换以表示插入目的字段的缺省值。除此以外，当VALUES出现在其他场合的时候是不能使用DEFAULT的。

- **sort_expression**

  一个表示如何排序结果行的表达式或者整数常量。

- **ASC**

  指定按照升序排列。

- **DESC**

  指定按照降序排列。

- **operator**

  一个排序操作符。

- **count**

  返回的最大行数。

- **start**

  开始返回行之前忽略的行数。

- **FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY**

  FETCH子句限定返回查询结果从第一行开始的总行数，count的缺省值为1。

示例

创建表reason_t1：

```
DROP TABLE IF EXISTS reason_t1;
CREATE TABLE reason_t1
(
   TABLE_SK          INTEGER               ,
   TABLE_ID          VARCHAR(20)           ,
   TABLE_NA          VARCHAR(20)
);
```

向表中插入一条记录：

```
INSERT INTO reason_t1(TABLE_SK, TABLE_ID, TABLE_NA) VALUES (1, 'S01', 'StudentA');
```

向表中插入一条记录，和上一条语法等效：

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA');
```

向表中插入TABLE_SK小于1的记录：

```
INSERT INTO reason_t1 SELECT * FROM reason_t1 WHERE TABLE_SK < 1;
```

向表中插入多条记录：

```
INSERT INTO reason_t1 VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB');
SELECT * FROM reason_t1 ORDER BY 1;
```

使用INSERT OVERWRITE更新表中的数据，即覆盖式插入数据：

```
INSERT OVERWRITE INTO reason_t1 values (4, 'S02', 'StudentB');
SELECT * FROM reason_t1 ORDER BY 1;
```

将表reason_t1的数据插入到表reason_t1中：

```
INSERT INTO reason_t1 SELECT * FROM reason_t1;
```

对独立的字段明确缺省值：

```
INSERT INTO reason_t1 VALUES (5, 'S03', DEFAULT);
```

将一个表中的部分数据插入到另一个表中：先通过WITH子查询得到一张临时表temp_t，然后将临时表temp_t中的所有数据插入另一张表reason_t1中：

```
WITH temp_t AS (SELECT * FROM reason_t1) INSERT INTO reason_t1 SELECT * FROM temp_t ORDER BY 1;
```


14 DCL 语法

14.1 DCL 语法一览表

DCL（Data Control Language数据控制语言），是用来设置或更改数据库用户或角色权限的语句。

授权

GaussDB(DWS)提供了针对数据对象和角色授权的语句，请参考**GRANT**。

收回权限

GaussDB(DWS)提供了收回权限的语句，请参考**REVOKE**。

设置默认权限

GaussDB(DWS)允许设置应用于将来创建的对象的权限，请参考**ALTER DEFAULT PRIVILEGES**。

14.2 ALTER DEFAULT PRIVILEGES

功能描述

设置应用于将来要创建的对象的权限（不会影响现有对象的权限）。

用户只可以修改由用户本身或者用户本身所属的角色所创建的对象的默认权限，这些权限可以对全局范围设置（即数据库中创建的所有对象），也可以为指定模式下的对象设置。

查看有关数据库用户的默认权限的信息，可以查询**PG_DEFAULT_ACL**系统表。

注意事项

目前只支持表（包括视图）、序列、函数和类型（包括域）的权限更改。

语法格式

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke;
```

- 其中abbreviated_grant_or_revoke子句用于指定对哪些对象进行授权或回收权限。

  ```
  grant_on_tables_clause
    | grant_on_functions_clause
    | grant_on_types_clause
    | grant_on_sequences_clause
    | revoke_on_tables_clause
    | revoke_on_functions_clause
    | revoke_on_types_clause
    | revoke_on_sequences_clause
  ```

- 其中grant_on_tables_clause子句用于对表授权。

  ```
  GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE |
  ANALYSE | VACUUM | ALTER | DROP }
      [, ...] | ALL [ PRIVILEGES ] }
      ON TABLES
      TO { [ GROUP ] role_name | PUBLIC } [, ...]
      [ WITH GRANT OPTION ]
  ```

- 其中grant_on_functions_clause子句用于对函数授权。

  ```
  GRANT { EXECUTE | ALL [ PRIVILEGES ] }
      ON FUNCTIONS
      TO { [ GROUP ] role_name | PUBLIC } [, ...]
      [ WITH GRANT OPTION ]
  ```

- 其中grant_on_types_clause子句用于对类型授权。

  ```
  GRANT { USAGE | ALL [ PRIVILEGES ] }
      ON TYPES
      TO { [ GROUP ] role_name | PUBLIC } [, ...]
      [ WITH GRANT OPTION ]
  ```

- 其中grant_on_sequences_clause子句用于对序列授权。

  ```
  GRANT { { USAGE | SELECT | UPDATE }
      [, ...] | ALL [ PRIVILEGES ] }
      ON SEQUENCES
      TO { [ GROUP ] role_name | PUBLIC } [, ...]
      [ WITH GRANT OPTION ]
  ```

- 其中revoke_on_tables_clause子句用于回收表对象的权限。

  ```
  REVOKE [ GRANT OPTION FOR ]
      { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE |
  ANALYSE | VACUUM | ALTER | DROP }
      [, ...] | ALL [ PRIVILEGES ] }
      ON TABLES
      FROM { [ GROUP ] role_name | PUBLIC } [, ...]
      [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
  ```

- 其中revoke_on_functions_clause子句用于回收函数的权限。

  ```
  REVOKE [ GRANT OPTION FOR ]
      { EXECUTE | ALL [ PRIVILEGES ] }
      ON FUNCTIONS
      FROM { [ GROUP ] role_name | PUBLIC } [, ...]
      [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
  ```

- 其中revoke_on_types_clause子句用于回收类型的权限。

  ```
  REVOKE [ GRANT OPTION FOR ]
      { USAGE | ALL [ PRIVILEGES ] }
      ON TYPES
      FROM { [ GROUP ] role_name | PUBLIC } [, ...]
      [ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
  ```

- 其中revoke_on_sequences_clause子句用于回收序列的权限。

  ```
  REVOKE [ GRANT OPTION FOR ]
      { { USAGE | SELECT | UPDATE }
  ```



```
[, ...] | ALL [ PRIVILEGES ] }
ON SEQUENCES
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT | CASCADE CONSTRAINTS ]
```

参数说明

- **target_role**

  已有角色的名称。如果省略FOR ROLE/USER，则缺省值为当前角色/用户。

  target_role必须有schema_name的CREATE权限。查看角色/用户是否具有schema的CREATE权限可使用has_schema_privilege函数。

  ```
  SELECT a.rolname, n.nspname FROM pg_authid as a, pg_namespace as n WHERE
  has_schema_privilege(a.oid, n.oid, 'CREATE');
  ```

  取值范围：已有角色的名称。

- **schema_name**

  现有模式的名称。如果指定了模式名，那么之后在这个模式下面创建的所有对象默认的权限都会被修改。如果IN SCHEMA被省略，那么全局权限会被修改。

  取值范围：现有模式的名称。

- **role_name**

  被授予或者取消权限角色的名称。

  取值范围：已存在的角色名称。

须知

如果想删除一个被赋予了默认权限的角色，有必要恢复改变的缺省权限或者使用DROP OWNED BY来为角色脱离缺省的权限记录。

示例

创建用户：

```
CREATE USER jack PASSWORD '{Password}';
```

创建模式：

```
CREATE SCHEMA tpcds;
```

- 将创建在模式tpcds里的所有表（和视图）的SELECT权限授予每一个用户：
  ```
  ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT SELECT ON TABLES TO PUBLIC;
  ```
- 将tpcds下的所有表的插入权限授予用户jack：
  ```
  ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds GRANT INSERT ON TABLES TO jack;
  ```
- 撤销上述权限：
  ```
  ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE SELECT ON TABLES FROM PUBLIC;
  ALTER DEFAULT PRIVILEGES IN SCHEMA tpcds REVOKE INSERT ON TABLES FROM jack;
  ```
- 假设有两个用户test1、test2，如果想要test2用户对test1用户未来创建的表都有查询权限可以用如下操作：
  - 创建用户test1、test2：
    ```
    CREATE USER test1 PASSWORD '{Password}';
    CREATE USER test2 PASSWORD '{Password}';
    ```
  - 首先，把test1的schema的权限赋权给test2用户：
    ```
    GRANT usage, create ON SCHEMA test1 TO test2;
    ```

- 其次，把test1用户下的表的查询权限赋值给test2用户：
  ```
  ALTER DEFAULT PRIVILEGES FOR USER test1 IN SCHEMA test1 GRANT SELECT ON tables TO test2;
  ```
- 然后，test1用户创建表：
  ```
  SET ROLE test1 password '{Password}';
  CREATE TABLE test3( a int, b int);
  ```
- 最后，用test2用户去查询：
  ```
  SET ROLE test2 password '{Password}';
  SELECT * FROM test1.test3;
  ```

相关链接

GRANT，**REVOKE**

14.3 ANALYZE | ANALYSE

功能描述

用于收集有关数据库中表内容的统计信息，统计结果存储在系统表PG_STATISTIC下。执行计划生成器会使用这些统计数据，以确定最有效的执行计划。

如果没有指定参数，ANALYZE会分析当前数据库中的每个表和分区表。同时也可以通过指定table_name、column和partition_name参数把分析限定在特定的表、列或分区表中。

能够执行ANALYZE特定表的用户，包括表的所有者、表所在数据库的所有者、通过GRANT被授予该表上ANALYZE权限的用户或者被授予了gs_role_analyze_any角色的用户以及有SYSADMIN属性的用户。

在百分比采样收集统计信息时，用户需要被授予ANALYZE和SELECT权限。

ANALYZE|ANALYSE VERIFY用于检测数据库中普通表（行存表、列存表）的数据文件是否损坏，目前此命令暂不支持HDFS表。

注意事项

- 仅8.1.1及以上集群版本支持在匿名块、事务块、函数或存储过程内对单表进行ANALYZE操作。
- 对于ANALYZE全库，库中各表的ANALYZE处于不同的事务中，所以不支持在匿名块、事务块、函数或存储过程内对全库执行ANALYZE。
- 统计信息的回滚操作不支持PG_CLASS中相关字段的回滚。
- ANALYZE VERIFY操作处理的大多为异常场景检测需要使用RELEASE版本。ANALYZE VERIFY场景不触发远程读，因此远程读参数不生效。对于关键系统表出现错误被系统检测出页面损坏时，将直接报错不再继续检测。

语法格式

- 收集表的统计信息。
  ```
  { ANALYZE | ANALYSE } [ VERBOSE ]
      [ table_name [ ( column_name [, ...] ) ] ];
  ```
- 收集分区表的统计信息。
  ```
  { ANALYZE | ANALYSE } [ VERBOSE ]
      [ table_name [ ( column_name [, ...] ) ] ]
      PARTITION ( patrition_name ) ;
  ```

说明

普通分区表目前支持针对某个分区的统计信息的语法，但功能上不支持针对某个分区的统计信息收集。对指定分区执行ANALYZE，会有相应的WARNING提示。

- 收集外表的统计信息。

```
{ ANALYZE | ANALYSE } [ VERBOSE ]
    { foreign_table_name | FOREIGN TABLES };
```

- 收集多列统计信息

```
{ANALYZE | ANALYSE} [ VERBOSE ]
    table_name (( column_1_name, column_2_name [, ...] ));
```

说明

- 收集多列统计信息时，请设置GUC参数default_statistics_target为负数，以使用百分比采样方式。
- 每组多列统计信息最多支持32列。
- 不支持收集多列统计信息的表：系统表、HDFS外表复制表。

- 检测当前库的数据文件

```
{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE};
```

说明

- 支持对全库进行操作，由于涉及的表较多，建议以重定向保存结果**gsql -d database -p port -f "verify.sql"> verify_warning.txt 2>&1**。
- 不支持HDFS表（内表和外表），不支持临时表和unlog表。
- 对外提示NOTICE只核对外可见的表，内部表的检测会包含在它所依赖的外部表，不对外显示和呈现。
- 此命令的处理可容错ERROR级别的处理。由于debug版本的Assert可能会导致core无法继续执行命令，建议在release模式下操作。
- 对于全库操作时，当关键系统表出现损坏则直接报错，不再继续执行。

- 检测表和索引的数据文件

```
{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name|index_name [CASCADE];
```

说明

- 支持对普通表的操作和索引表的操作，但不支持对索引表index使用CASCADE操作。原因是CASCADE模式用于处理主表的所有索引表，当单独对索引表进行检测时，无需使用CASCADE模式。
- 不支持HDFS表（内表和外表），不支持临时表和unlog表。
- 对于主表的检测会同步检测主表的内部表，例如toast表、cudesc表等。
- 当提示索引表损坏时，建议使用reindex命令进行重建索引操作。

- 检测表分区的数据文件

```
{ANALYZE | ANALYSE} VERIFY {FAST|COMPLETE} table_name PARTITION {(patrition_name)}[CASCADE];
```

说明

- 支持对表的单独分区进行检测操作，但不支持对索引表index使用CASCADE操作。
- 不支持HDFS表（内表和外表），不支持临时表和unlog表.

参数说明

- **VERBOSE**

  启用显示进度信息。

说明

如果指定了VERBOSE，ANALYZE发出进度信息，表明目前正在处理的表。各种有关表的统计信息也会打印出来。

- **table_name**

  需要分析的特定表的表名（可能会带模式名），如果省略，将对数据库中的所有表（非外部表）进行分析。

  对于ANALYZE收集统计信息，目前仅支持行存表、列存表、HDFS表、ORC格式的OBS外表、CARBONDATA格式的OBS外表、协同分析的外表。

  取值范围：已有的表名。

- **column_name**，column_1_name，column_2_name

  需要分析特定列的列名，默认为所有列。

  取值范围：已有的列名。

- **partition_name**

  如果table为分区表，在关键字PARTITION后面指定分区名partition_name表示分析该分区表的统计信息。目前语法上支持分区表做ANALYZE，但功能实现上暂不支持对指定分区统计信息的分析。

  取值范围：表的某一个分区名。

- **foreign_table_name**

  需要分析的特定外表的表名（可能会带模式名），该表的数据存放于HDFS分布式文件系统中。

  取值范围：已有的表名。

- **FOREIGN TABLES**

  分析所有当前用户权限下，数据位于HDFS分布式文件系统中的HDFS外表。

- **index_name**

  需要分析的特定索引表的表名（可能会带模式名）。

  取值范围：已有的表名。

- **FAST|COMPLETE**

  对于行存表，FAST模式下主要对于行存表的CRC和page header进行校验，如果校验失败则会告警; 而COMPLETE模式下，则主要对行存表的指针、tuple进行解析校验。 对于列存表，FAST模式下主要对于列存表的CRC和magic进行校验，如果校验失败则会告警；而COMPLETE模式下，则主要对列存表的CU进行解析校验。

- **CASCADE**

  CASCADE模式下会对当前表的所有索引进行检测处理。

示例

```
DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE CUSTOMER
(
   C_CUSTKEY     BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY  ,
   C_NAME        VARCHAR(25)  ,
   C_ADDRESS     VARCHAR(40)  ,
   C_NATIONKEY   INT          ,
   C_PHONE       CHAR(15)     ,
   C_ACCTBAL     DECIMAL(15,2)
)
DISTRIBUTE BY HASH(C_CUSTKEY);
```

使用ANALYZE语句更新表customer_info统计信息：


```
ANALYZE CUSTOMER;
```

使用ANALYZE VERBOSE语句更新表customer_info统计信息，并输出表的相关信息：

```
ANALYZE VERBOSE CUSTOMER;
```

14.4 DEALLOCATE

功能描述

DEALLOCATE用于删除先前编写的预备语句。如果未显式删除一个预备语句，则在会话结束时将其删除。

有关预备语句可参考**PREPARE**。

注意事项

无。

语法格式

```
DEALLOCATE [ PREPARE ] { name | ALL };
```

参数说明

- **PREPARE**

  可选关键字，总被忽略。

- **name**

  将要删除的预备语句。

- **ALL**

  删除所有预备语句。

示例

无。

14.5 DO

功能描述

执行匿名代码块。

代码块被视为没有参数的函数主体，返回值类型是void。它的解析和执行是同一时刻发生的。

注意事项

- 程序语言在使用之前，必须通过命令CREATE LANGUAGE安装到当前的数据库中。 plpgsql是默认的安装语言，其它语言安装时必须指定。
- 如果语言是不受信任的，用户必须有使用程序语言的USAGE权限，或者是系统管理员。

语法格式

```
DO [ LANGUAGE lang_name ] code;
```

参数说明

- **lang_name**

  用来解析代码的程序语言的名字，如果缺省，默认的语言是plpgsql。

- **code**

  程序语言代码可以被执行的。程序语言必须指定为字符串才行。

示例

授予用户webuser对模式tpcds下视图的所有操作权限：

```
DO $$DECLARE r record;
BEGIN
   FOR r IN SELECT c.relname,n.nspname FROM pg_class c,pg_namespace n
         WHERE c.relnamespace = n.oid AND n.nspname = 'tpcds' AND relkind IN ('r','v')
   LOOP
      EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || ' TO
webuser';
   END LOOP;
END$$;
```

14.6 EXECUTE

功能描述

用来执行一个前面准备好的预备语句。由于预备语句只在会话的生命期里存在，那么该预备语句必须在当前会话中由一个更早执行的PREPARE语句创建。

注意事项

如果创建预备语句的PREPARE语句声明了一些参数，那么传递给EXECUTE语句的必须是一个兼容的参数集，否则就会生成一个错误。

语法格式

```
EXECUTE name [ ( parameter [, ...] ) ];
```

参数说明

- **name**

  要执行的预备语句的名字。

- **parameter**

  给预备语句的一个参数的具体数值。它必须是一个生成与创建这个预备语句时指定参数的数据类型相兼容的值的表达式。

示例

为一个INSERT语句创建一个预备语句，然后执行它：

```
CREATE TABLE IF NOT EXISTS reason_t1 (a int,b varchar(20),c varchar(20));
PREPARE insert_reason(integer,character(16),character(100)) AS INSERT INTO reason_t1 VALUES($1,$2,$3);
EXECUTE insert_reason(52, 'AAAAAAAADDAAAAAA', 'reason 52');
```


14.7 EXECUTE DIRECT

功能描述

在指定的节点上执行SQL语句。一般情况下，SQL语句的执行是由集群负载自动分配到合适的节点上，execute direct主要用于数据库维护和测试。

注意事项

- 只有系统管理员才能执行EXECUTE DIRECT。
- 为了各个节点上数据的一致性，SQL语句仅支持SELECT，不允许执行事务语句、DDL、DML。
- 使用此类型语句在指定的DN执行AVG聚集计算时，返回结果集是以数组形式返回，如{4,2}，表示sum结果为4，count结果为2。
- 由于CN节点不存储用户表数据，没有必要指定CN节点执行用户表上的SELECT查询。如果查询语句中含有隐藏的远程调用开启事务逻辑，则会报错“Cannot send begin to other nodes as I'm not execute cn”。
- 不允许执行嵌套的EXECUTE DIRECT语句，即执行的SQL语句不能同样是EXECUTE DIRECT语句，此时可直接执行最内层EXECUTE DIRECT语句代替。

语法格式

```
EXECUTE DIRECT ON ( nodename [, ... ] ) query ;
```

参数说明

- **nodename**

  节点名称。

  取值范围：已存在的节点。

- **query**

  要执行查询语句。

示例

查询节点dn_6001_6002上tpcds.customer_address记录：

```
CREATE TABLE IF NOT EXISTS customer_address(a int,b int);
EXECUTE DIRECT ON(dn_6001_6002) 'select count(*) from customer_address';
```

14.8 GRANT

功能描述

对角色和用户进行授权操作。

使用GRANT命令进行用户授权包括以下三种场景：

- **将系统权限授权给角色或用户**

系统权限又称为用户属性，包括SYSADMIN、CREATEDB、CREATEROLE、AUDITADMIN和LOGIN。

系统权限一般通过CREATE/ALTER ROLE语法来指定。其中，SYSADMIN权限可以通过GRANT/REVOKE ALL PRIVILEGE授予或撤销。但系统权限无法通过ROLE和USER的权限被继承，也无法授予PUBLIC。

- **将数据库对象授权给角色或用户**

  将数据库对象（表和视图、指定字段、数据库、函数、模式等）的相关权限授予特定角色或用户。

  GRANT命令将数据库对象的特定权限授予一个或多个角色。这些权限会追加到已有的权限上。

  关键字PUBLIC表示该权限要赋予所有角色，包括以后创建的用户。PUBLIC可以看做是一个隐含定义好的组，它总是包括所有角色。任何角色或用户都将拥有通过GRANT直接赋予的权限和所属的权限，再加上PUBLIC的权限。

  如果声明了WITH GRANT OPTION，则被授权的用户也可以将此权限赋予他人，否则就不能授权给他人。这个选项不能赋予PUBLIC，这是GaussDB(DWS)特有的属性。

  GaussDB(DWS)会将某些类型的对象上的权限授予PUBLIC。默认情况下，对表、表字段、序列、外部数据源、外部服务器、模式或表空间对象的权限不会授予PUBLIC，而以下这些对象的权限会授予PUBLIC：数据库的CONNECT权限和CREATE TEMP TABLE权限、函数的EXECUTE特权、语言和数据类型（包括域）的USAGE特权。对象拥有者可以撤销默认授予PUBLIC的权限并专门授予权限给其他用户。为了更安全，建议在同一个事务中创建对象并设置权限，这样其他用户就没有时间窗口使用该对象。另外，这些初始的默认权限可以使用ALTER DEFAULT PRIVILEGES命令修改。

- **将角色或用户的权限授权给其他角色或用户**

  将一个角色或用户的权限授予一个或多个其他角色或用户。在这种情况下，每个角色或用户都可视为拥有一个或多个数据库权限的集合。

  当声明了WITH ADMIN OPTION，被授权的用户可以将该权限再次授予其他角色或用户，以及撤销所有由该角色或用户继承到的权限。当授权的角色或用户发生变更或被撤销时，所有继承该角色或用户权限的用户拥有的权限都会随之发生变更。

  数据库系统管理员可以给任何角色或用户授予/撤销任何权限。拥有CREATEROLE权限的角色可以赋予或者撤销任何非系统管理员角色的权限。

注意事项

无。

语法格式

- 将表或视图的访问权限赋予指定的用户或角色。不允许对表分区进行GRANT操作，对表分区进行GRANT操作会引起告警。

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE |
ANALYSE | VACUUM | ALTER | DROP } [, ...]
    | ALL [ PRIVILEGES ] }
  ON { [ TABLE ] table_name [, ...]
    | ALL TABLES IN SCHEMA schema_name [, ...] }
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将表中字段的访问权限赋予指定的用户或角色。

```
GRANT { {{ SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )} [, ...]
    | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
```



```
  ON [ TABLE ] table_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将数据库的访问权限赋予指定的用户或角色。

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
    | ALL [ PRIVILEGES ] }
  ON DATABASE database_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将域的访问权限赋予指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON DOMAIN domain_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

说明

当前版本暂时不支持赋予域的访问权限。

- 将外部数据源的访问权限赋予给指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON FOREIGN DATA WRAPPER fdw_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将外部服务器的访问权限赋予给指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON FOREIGN SERVER server_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将函数的访问权限赋予给指定的用户或角色。

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON { FUNCTION {function_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )} [, ...]
    | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将过程语言的访问权限赋予给指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON LANGUAGE lang_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

说明

当前版本暂时不支持过程语言。

- 将大对象的访问权限赋予指定的用户或角色。

```
GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
  ON LARGE OBJECT loid [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

说明

当前版本暂时不支持大对象。

- 将序列的访问权限赋予指定的用户或角色。

```
GRANT { { SELECT | UPDATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
  ON { SEQUENCE sequence_name [, ...]
      | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将子集群的访问权限赋予指定的用户或角色。普通用户不能执行针对Node Group的GRANT/REVOKE操作。

```
GRANT { CREATE | USAGE | COMPUTE | ALL [ PRIVILEGES ] }
  ON NODE GROUP group_name [, ...]
```

```
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将模式的访问权限赋予指定的用户或角色。

```
GRANT { { CREATE | USAGE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
  ON SCHEMA schema_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

  说明

  将模式中的表或者视图对象授权给其他用户时，需要将表或视图所属的模式的USAGE权限同时授予该用户，若没有该权限，则只能看到这些对象的名字，并不能实际进行对象访问。

- 将类型的访问权限赋予指定的用户或角色。

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON TYPE type_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

  说明

  当前版本暂时不支持赋予类型的访问权限。

- 将角色的权限赋予其他用户或角色的语法。

```
GRANT role_name [, ...]
  TO role_name [, ...]
  [ WITH ADMIN OPTION ];
```

- 将sysadmin权限赋予指定的角色。

```
GRANT ALL { PRIVILEGES | PRIVILEGE }
  TO role_name;
```

参数说明

GRANT的权限分类如下所示。

- **SELECT**

  允许对指定的表、视图、序列执行SELECT语句。

- **INSERT**

  允许对指定的表执行INSERT语句。

- **UPDATE**

  允许对声明的表中任意字段执行UPDATE语句。SELECT… FOR UPDATE和SELECT… FOR SHARE除了需要SELECT权限外，还需要UPDATE权限。

- **DELETE**

  允许执行DELETE语句删除指定表中的数据。

- **TRUNCATE**

  允许执行TRUNCATE语句删除指定表中的所有记录。

- **REFERENCES**

  创建一个外键约束，必须拥有参考表和被参考表的REFERENCES权限。

- **TRIGGER**

  创建一个触发器，必须拥有表或视图的TRIGGER权限。

- **ANALYZE | ANALYSE**

  对表执行ANALYZE | ANALYSE操作来收集表的统计信息，必须拥有表的ANALYZE | ANALYSE权限。

- **CREATE**
  - 对于数据库，允许在数据库里创建新的模式。
  - 对于模式，允许在模式中创建新的对象。如果要重命名一个对象，用户除了必须是该对象的所有者外，还必须拥有该对象所在模式的CREATE权限。
  - 对于子集群，允许在子集群中创建表对象。
- **CONNECT**

  允许用户连接到指定的数据库。
- **TEMPORARY | TEMP**

  允许在使用指定数据库时创建临时表。
- **EXECUTE**

  允许使用指定的函数，以及利用这些函数实现的操作符。
- **USAGE**
  - 对于过程语言，允许用户在创建函数的时候指定过程语言。
  - 对于模式，USAGE允许访问包含在指定模式中的对象，若没有该权限，则只能看到这些对象的名字。
  - 对于序列，USAGE允许使用nextval函数。
  - 对于子集群， 对包含在指定模式中的对象有访问权限时，USAGE允许访问指定子集群下的表对象。
- **COMPUTE**

  针对计算子集群，允许用户在具有compute权限的计算子集群上进行弹性计算。
- **ALL PRIVILEGES**

  一次赋予指定用户/角色所有可赋予的权限。只有系统管理员有权执行GRANT ALL PRIVILEGES。
- **WITH GRANT OPTION**

  指定权限是否允许转授。如果声明了WITH GRANT OPTION，则被授权的用户也可以将此权限赋予他人，否则就不能授权给他人。这个选项不能赋予PUBLIC。

  说明

  - NODE GROUP不支持WITH GRANT OPTION功能。
  - 使用with grant option时需确保enable_grant_option参数设置为on。
- **WITH ADMIN OPTION**

  指定权限是否允许转授。如果声明了WITH ADMIN OPTION，角色的成员又可以将角色的成员身份授予其他人。

GRANT的参数说明如下所示。

- **role_name**

  已存在用户名称。
- **table_name**

  已存在表名称。
- **column_name**

  已存在字段名称。
- **schema_name**

  已存在模式名称。

- **database_name**
  已存在数据库名称。
- **function_name**
  已存在函数名称。
- **sequence_name**
  已存在序列名称。
- **domain_name**
  已存在域类型名称。
- **fdw_name**
  已存在外部数据包名称。
- **lang_name**
  已存在语言名称。
- **type_name**
  已存在类型名称。
- **group_name**
  已存在的子集群名称。
- **argmode**
  参数模式。
  取值范围：字符串，要符合标识符命名规范。
- **arg_name**
  参数名称。
  取值范围：字符串，要符合标识符命名规范。
- **arg_type**
  参数类型。
  取值范围：字符串，要符合标识符命名规范。
- **loid**
  包含本页的大对象的标识符。
  取值范围：字符串，要符合标识符命名规范。

示例

创建用户：

```
CREATE USER joe PASSWORD '{Password}';
CREATE USER kim PASSWORD '{Password}';
```

创建模式：

```
CREATE SCHEMA tpcds;
```

创建表：

```
CREATE TABLE IF NOT EXISTS tpcds.reason(r_reason_sk int,r_reason_id int,r_reason_desc int);
```

- **将系统权限授权给用户或者角色。**
  - 将sysadmin所有可用权限授权给joe用户：
    ```
    GRANT ALL PRIVILEGES TO joe;
    ```

授权成功后，用户joe会拥有sysadmin的所有权限。

- **将对象权限授权给用户或者角色。**
  - 将表tpcds.reason的SELECT权限授权给用户joe：
    ```
    GRANT SELECT ON TABLE tpcds.reason TO joe;
    ```
  - 将表tpcds.reason的所有权限授权给用户kim：
    ```
    GRANT ALL PRIVILEGES ON tpcds.reason TO kim;
    ```
    授权成功后，kim用户就拥有了tpcds.reason表的所有权限，包括增删改查等权限。
  - 将模式tpcds的使用权限授权给用户joe：
    ```
    GRANT USAGE ON SCHEMA tpcds TO joe;
    ```
    授权成功后，joe用户就拥有了模式schema的USAGE权限，允许访问包含在指定模式schema中的对象。
  - 将tpcds.reason表中r_reason_sk、r_reason_id、r_reason_desc列的查询权限，r_reason_desc的更新权限授权给joe：
    ```
    GRANT select (r_reason_sk,r_reason_id,r_reason_desc),update (r_reason_desc) ON tpcds.reason TO joe;
    ```
    授权成功后，用户joe对tpcds.reason表中r_reason_sk，r_reason_id的查询权限会立即生效。
    ```
    GRANT select (r_reason_sk, r_reason_id) ON tpcds.reason TO joe ;
    ```
  - 将函数func_add_sql的EXECUTE权限授权给用户joe。
    ```
    CREATE FUNCTION func_add_sql(integer, integer) RETURNS integer
        AS 'select $1 + $2;'
        LANGUAGE SQL
        IMMUTABLE
        RETURNS NULL ON NULL INPUT;
    GRANT EXECUTE ON FUNCTION func_add_sql(integer, integer) TO joe;
    ```
  - 将序列serial的UPDATE权限授权给joe用户。
    ```
    CREATE SEQUENCE serial START 101 CACHE 20;
    GRANT UPDATE ON SEQUENCE serial TO joe;
    ```
  - 将数据库gaussdb的连接权限授权给用户joe，并给予其在gaussdb中创建schema的权限：
    ```
    GRANT create,connect on database gaussdb TO joe ;
    ```
  - 将模式tpcds的访问权限授权给角色joe，并授予该角色在tpcds下创建对象的权限，不允许该角色中的用户将权限授权给其他人：
    ```
    GRANT USAGE,CREATE ON SCHEMA tpcds TO joe;
    ```
- **将用户或者角色的权限授权给其他用户或角色。**
  - 将用户joe的权限授权给用户kim，并允许该角色将权限授权给其他人：
    ```
    GRANT joe TO kim WITH ADMIN OPTION;
    ```
  - 将用户joe的权限授权给kim用户：
    ```
    GRANT joe TO kim;
    ```

相关链接

REVOKE，**ALTER DEFAULT PRIVILEGES**

14.9 PREPARE

功能描述

创建一个预备语句。

预备语句是服务端的对象，可以用于优化性能。在执行PREPARE语句的时候，指定的查询被解析、分析、重写。当随后发出EXECUTE语句的时候，预备语句被规划和执行。这种设计避免了重复解析、分析工作。PREPARE语句创建后在整个数据库会话期间一直存在，一旦创建成功，即便是在事务块中创建，事务回滚，PREPARE也不会删除。只能通过显式调用**DEALLOCATE**进行删除，会话结束时，PREPARE也会自动删除。

注意事项

无。

语法格式

```
PREPARE name [ ( data_type [, ...] ) ] AS statement;
```

参数说明

- **name**
  指定预备语句的名称。该名称必须在该会话中是唯一的。
- **data_type**
  参数的数据类型。
- **statement**
  任何SELECT、INSERT、UPDATE、DELETE或VALUES语句。

示例

为一个INSERT语句创建一个预备语句，然后执行它：

```
CREATE TABLE IF NOT EXISTS reason_t1 (a int,b varchar(20),c varchar(20));
PREPARE insert_reason(integer,character(16),character(100)) AS INSERT INTO reason_t1 VALUES($1,$2,$3);
EXECUTE insert_reason(52, 'AAAAAAAADDAAAAAA', 'reason 52');
```

相关链接

DEALLOCATE

14.10 REASSIGN OWNED

功能描述

修改数据库对象的属主。

REASSIGN OWNED要求系统将所有old_role拥有的数据库对象的属主更改为new_role。

注意事项

- REASSIGN OWNED常用于在删除角色之前的准备工作。由于不会影响其他数据库中的对象， 因此通常需要在每个数据库中执行此命令，该数据库包含要删除的角色所拥有的对象。
- 执行REASSIGN OWNED需要有原角色和目标角色上的权限。
- 资源管理不对该语法的数据切换进行监控，需调用select gs_wlm_readjust_user_space(0)手动校准监控数据。

语法格式

```
REASSIGN OWNED BY old_role [, ...] TO new_role;
```

参数说明

- **old_role**
  旧属主的角色名。
- **new_role**
  将要成为这些对象属主的新角色的名字。

示例

创建用户：

```
CREATE USER joe PASSWORD '{Password}';
CREATE USER jack PASSWORD '{Password}';
```

将由joe和jack角色拥有的所有数据库对象重新分配给admin：

```
REASSIGN OWNED BY joe, jack TO dbadmin;
```

14.11 REVOKE

功能描述

REVOKE用于撤销一个或多个角色的权限。

注意事项

非对象所有者试图在对象上REVOKE权限，命令按照以下规则执行：

- 如果授权用户没有该对象上的权限，则命令立即失败。
- 如果授权用户有部分权限，则只撤销那些有授权选项的权限。
- 如果授权用户没有授权选项，REVOKE ALL PRIVILEGES形式将发出一个错误信息，而对于其他形式的命令而言，如果是命令中指定名字的权限没有相应的授权选项，该命令将发出一个警告。
- 不允许对表分区进行REVOKE操作，对分区表进行REVOKE操作会引起告警。

语法格式

- 撤销指定表和视图上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER | ANALYZE |
ANALYSE | VACUUM | ALTER | DROP }[, ...]
    | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
        | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销表上指定字段权限。

```
REVOKE [ GRANT OPTION FOR ]
    { {{ SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )}[, ...]
    | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
```

```
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销指定数据库上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
    | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销指定函数上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION {function_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )} [, ...]
        | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销指定大对象上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销指定序列上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCE sequence_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销指定模式上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE | ALTER | DROP } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销指定子集群上权限。

```
REVOKE [ GRANT OPTION FOR ]
    { CREATE | USAGE | COMPUTE | ALL [ PRIVILEGES ] }
    ON NODE GROUP group_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```

- 按角色撤销角色上的权限。

```
REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_name [, ...]
    [ CASCADE | RESTRICT ];
```

- 撤销角色上的sysadmin权限。

```
REVOKE ALL { PRIVILEGES | PRIVILEGE } FROM role_name;
```

参数说明

关键字PUBLIC表示一个隐式定义的拥有所有角色的组。

权限类别和参数说明，请参见GRANT的**参数说明**。

任何特定角色拥有的特权包括直接授予该角色的特权、从该角色作为其成员的角色中得到的权限以及授予给PUBLIC的权限。因此，从PUBLIC收回SELECT特权并不一定会意味着所有角色都会失去在该对象上的SELECT特权，那些直接被授予的或者通过另一个角色被授予的角色仍然会拥有它。类似地，从一个用户收回SELECT后，如果PUBLIC仍有SELECT权限，该用户还是可以使用SELECT。

指定GRANT OPTION FOR时，只撤销对该权限授权的权力，而不撤销该权限本身。

如用户A拥有某个表的UPDATE权限，及WITH GRANT OPTION选项，同时A把这个权限赋予了用户B，则用户B持有的权限称为依赖性权限。当用户A持有的权限或者授权选项被撤销时，依赖性权限仍然存在，但如果声明了CASCADE，则所有依赖性权限都被撤销。

一个用户只能撤销由它自己直接赋予的权限。例如，如果用户A被指定授权（WITH ADMIN OPTION）选项，且把一个权限赋予了用户B，然后用户B又赋予了用户C，则用户A不能直接将C的权限撤销。但是，用户A可以撤销用户B的授权选项，并且使用CASCADE。这样，用户C的权限就会自动被撤销。另外一个例子：如果A和B都赋予了C同样的权限，则A可以撤销他自己的授权选项，但是不能撤销B的，因此C仍然拥有该权限。

如果执行REVOKE的角色持有的权限是通过多层成员关系获得的，则具体是哪个包含的角色执行的该命令是不确定的。在这种场合下，最好的方法是使用SET ROLE成为特定角色，然后执行REVOKE，否则可能导致删除了不想删除的权限，或者是任何权限都没有删除。

示例

创建用户jim：

```
CREATE USER jim PASSWORD '{Password}';
```

创建模式：

```
CREATE SCHEMA tpcds;
```

创建数据库：

```
CREATE DATABASE mydatabase OWNER jim;
```

创建表：

```
CREATE TABLE IF NOT EXISTS tpcds.reason(r_reason_sk int,r_reason_id int,r_reason_desc int);
```

创建视图：

```
CREATE VIEW myview AS select * from tpcds.reason;
```

撤销jim用户的所有权限：

```
REVOKE ALL PRIVILEGES FROM jim;
```

撤销指定模式上授予的权限：

```
REVOKE USAGE,CREATE ON SCHEMA tpcds FROM jim;
```

撤销jim用户的CONNECT特权：

```
REVOKE CONNECT ON DATABASE mydatabase FROM jim;
```

从用户jim撤销角色dbadmin中的成员资格：

```
REVOKE dbadmin FROM jim;
```

撤销用户jim对视图myView具有的所有特权：

```
REVOKE ALL PRIVILEGES ON myView FROM jim;
```

撤销针对表customer_t1的公共插入特权：

```
REVOKE INSERT ON tpcds.reason FROM PUBLIC;
```

撤销用户jim对tpcds.reason表中r_reason_sk，r_reason_id的查询权限：

```
REVOKE select (r_reason_sk, r_reason_id) ON tpcds.reason FROM jim;
```

撤销用户jim的函数权限：

```
CREATE FUNCTION func_add_sql(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
REVOKE execute ON FUNCTION func_add_sql(integer, integer) FROM jim CASCADE;
```

相关链接

GRANT

15 DQL 语法

15.1 DQL 语法一览表

DQL（ Data Query Language数据查询语言），用于从表或视图中获取数据。

查询

GaussDB(DWS)提供了用于从表或视图中获取数据的语句，请参考**SELECT**。

根据查询结果定义一个新表。

GaussDB(DWS)提供了根据查询结果创建一个新表，并且将查询到的数据插入到新表中的语句，请参考**SELECT INTO**。

15.2 SELECT

功能描述

SELECT用于从表或视图中读取数据。

SELECT语句就像叠加在数据库表上的过滤器，利用SQL关键字从数据表中过滤出用户需要的数据。

注意事项

- SELECT支持普通表和HDFS的Join，不支持普通表和GDS外表的join。即SELECT语句中不能同时出现普通表和GDS外表。
- 必须对每个在SELECT命令中使用的字段有SELECT权限。
- 使用FOR UPDATE或FOR SHARE还要求UPDATE权限。

语法格式

- 查询数据

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
```



```
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW {window_name AS ( window_definition )} [, ...] ]
[ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] | nlssort_expression_clause ] [ NULLS { FIRST |
LAST } ]} [, ...] ]
[ { [ LIMIT { count | ALL } ] [ OFFSET start [ ROW | ROWS ] ] } | { LIMIT start, { count | ALL } } ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ]} [...] ];
```

说明

condition和expression中可以使用targetlist中表达式的别名。

- 只能同一层引用。
- 只能引用targetlist中的别名。
- 只能是后面的表达式引用前面的表达式。
- 不能包含volatile函数。
- 不能包含Window function函数。
- 不支持在join on条件中引用别名。
- targetlist中有多个要应用的别名则报错。

- 其中子查询with_query为：

```
with_query_name [ ( column_name [, ...] ) ]
    AS [ [ NOT ] MATERIALIZED ] ( {select | values | insert | update | delete} )
```

- 其中指定查询源from_item为：

```
{[ ONLY ] table_name [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
|with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ]
|function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
|from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]}
```

- 其中group子句为：

```
( )
| expression
| ( expression [, ...] )
| ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
| CUBE ( { expression | ( expression [, ...] ) } [, ...] )
| GROUPING SETS ( grouping_element [, ...] )
```

- 其中指定分区partition_clause为：

```
PARTITION { ( partition_name ) |
      FOR (  partition_value [, ...] ) }
```

说明

指定分区只适合普通表。

- 其中设置排序方式nlssort_expression_clause为：

```
NLSSORT ( column_name, ' NLS_SORT = { SCHINESE_PINYIN_M | generic_m_ci } ' )
```

- 简化版查询语法，功能相当于select * from table_name。

```
TABLE { ONLY {(table_name)| table_name} | table_name [ * ]};
```

参数说明

- **WITH [RECURSIVE] with_query [, ...]**

  用于声明一个或多个可以在主查询中通过名字引用的子查询，相当于临时表。

  如果声明了RECURSIVE，那么允许SELECT子查询通过名字引用它自己。

其中with_query的详细格式为：with_query_name [(column_name [, ...])] AS [[NOT] MATERIALIZED] ({select | values | insert | update | delete})

- with_query_name指定子查询生成的结果集名字，在查询中可使用该名称访问子查询的结果集。
- 默认情况下，被主查询多次引用的with_query通常只被执行一次，并将其结果集进行物化，供主查询多次查询其结果集；被主查询引用一次的with_query，则不再单独执行，而是将其子查询直接替换到主查询中的引用处，随主查询一起执行。显示指定[NOT] MATERIALIZED，可改变默认行为：
  - 指定MATERIALIZED时，将子查询执行一次，并将其结果集进行物化。
  - 指定NOT MATERIALIZED时，则将其子查询替换到主查询中的引用处。以下几种情况会忽略NOT MATERIALIZED：
    - 子查询中含有volatile函数。
    - 子查询为含有FOR UPDATE/FOR SHARE的SELECT/VALUES语句。
    - 子查询为INSERT/UPDATE/DELETE等语句。
    - with_query为RECURSIVE。
    - 被引用次数大于1的with_query2引用了外层自引用的with_query1，则with_query2不能被替换到引用处。

      例如下面示例中，tmp2被引用了两次，tmp2因为引用了外层自引用的tmp1，所以即使tmp2指定了NOT MATERIALIZED也会被物化。

      ```
      with recursive tmp1(b) as (values(1)
      union all
      (with tmp2 as not materialized (select * from tmp1)
       select tt1.b + tt2.b from tmp2 tt1, tmp2 tt2))
      select * from tmp1;
      ```
- column_name指定子查询结果集中显示的列名。
- 每个子查询可以是SELECT，VALUES，INSERT，UPDATE或DELETE语句。

- **plan_hint子句**

  以/*+ */的形式在SELECT关键字后，用于对SELECT对应的语句块生成的计划进行hint调优，详细用法请参见章节：使用Plan Hint进行调优。

- **ALL**

  声明返回所有符合条件的行，是默认行为，可以省略该关键字。

- **DISTINCT [ON (expression [, ...])]**

  从SELECT的结果集中删除所有重复的行，使结果集中的每行都是唯一的。

  ON (expression [, ...]) 只保留那些在给出的表达式上运算出相同结果的行集合中的第一行。

  须知

  DISTINCT ON表达式是使用与ORDER BY相同的规则进行解释的。除非使用了ORDER BY来保证需要的行首先出现，否则，"第一行" 是不可预测的。

- **SELECT列表**

  指定查询表中列名，可以是部分列或者是全部（使用通配符*表示）。

通过使用子句AS output_name可以为输出字段取个别名，这个别名通常用于输出字段的显示。

列名可以用下面几种形式表达：

- 手动输入列名，多个列之间用英文逗号（,）分隔。
- 可以是FROM子句里面计算出来的字段。

- **FROM子句**

  为SELECT声明一个或者多个源表。

  FROM子句涉及的元素如下所示。

  - table_name

    表名或视图名，名称前可加上模式名，如：schema_name.table_name。

  - alias

    给表或复杂的表引用起一个临时的表别名，以便被其余的查询引用。

    别名用于缩写或者在自连接中消除歧义。如果提供了别名，它就会完全隐藏表的实际名字。

  - column_alias

    列别名

  - PARTITION

    查询分区表的某个分区的数据。

  - partition_name

    分区名。

  - partition_value

    指定的分区键值。在创建分区表时，如果指定了多个分区键，可以通过PARTITION FOR子句指定的这一组分区键的值，唯一确定一个分区。

  - subquery

    FROM子句中可以出现子查询，创建一个临时表保存子查询的输出。

  - with_query_name

    WITH子句同样可以作为FROM子句的源，可以通过WITH查询的名字对其进行引用。

  - function_name

    函数名称。函数调用也可以出现在FROM子句中。

  - join_type

    有5种类型，如下所示。

    - [INNER] JOIN

      一个JOIN子句组合两个FROM项。可使用圆括弧以决定嵌套的顺序。如果没有圆括弧，JOIN从左向右嵌套。

      在任何情况下，JOIN都比逗号分隔的FROM项绑定得更紧。

    - LEFT [OUTER] JOIN

      返回笛卡尔积中所有符合连接条件的行，再加上左表中通过连接条件没有匹配到右表行的那些行。这样，左边的行将扩展为生成表的全长，方法是在那些右表对应的字段位置填上NULL。请注意，只在计算匹配的时候，才使用JOIN子句的条件，外层的条件是在计算完毕之后施加的。

- RIGHT [OUTER] JOIN

  返回所有内连接的结果行，加上每个不匹配的右边行（左边用NULL扩展）。

  这只是一个符号上的方便，因为总是可以把它转换成一个LEFT OUTER JOIN，只要把左边和右边的输入互换位置即可。

- FULL [OUTER] JOIN

  返回所有内连接的结果行，加上每个不匹配的左边行（右边用NULL扩展），再加上每个不匹配的右边行（左边用NULL扩展）。

- CROSS JOIN

  CROSS JOIN等效于INNER JOIN ON（TRUE），即没有被条件删除的行。这种连接类型只是符号上的方便，因为它们与简单的FROM和WHERE的效果相同。

  说明

  必须为INNER和OUTER连接类型声明一个连接条件，即NATURAL ON，join_condition，USING (join_column [， ...]) 之一。但是它们不能出现在CROSS JOIN中。

其中CROSS JOIN和INNER JOIN生成一个简单的笛卡尔积，和在FROM的顶层列出两个项的结果相同。

– ON join_condition

  连接条件，用于限定连接中的哪些行是匹配的。如：ON left_table.a = right_table.a。

– USING(join_column[，...])

  ON left_table.a = right_table.a AND left_table.b = right_table.b ... 的简写。要求对应的列必须同名。

– NATURAL

  NATURAL是具有相同名称的两个表的所有列的USING列表的简写。

– from item

  用于连接的查询源对象的名称。

- **WHERE子句**

  WHERE子句构成一个行选择表达式，用来缩小SELECT查询的范围。condition是返回值为布尔型的任意表达式，任何不满足该条件的行都不会被检索。

  WHERE子句中可以通过指定"(+)"操作符的方法将表的连接关系转换为外连接。但是不建议用户使用这种用法，因为这并不是SQL的标准语法，在做平台迁移的时候可能面临语法兼容性的问题。同时，使用"(+)"有很多限制：

  a. "(+)"只能出现在where子句中。
  b. 如果from子句中已经有指定表连接关系，那么不能再在where子句中使用"(+)"。
  c. "(+)"只能作用在表或者视图的列上，不能作用在表达式上。
  d. 如果表A和表B有多个连接条件，那么必须在所有的连接条件中指定"(+)"，否则"(+)"将不会生效，表连接会转化成内连接，并且不给出任何提示信息。
  e. "(+)"作用的连接条件中的表不能跨查询或者子查询。如果"(+)"作用的表，不在当前查询或者子查询的from子句中，则会报错。如果"(+)"作用的对端的表不存在，则不报错，同时连接关系会转化为内连接。

f. "(+)"作用的表达式不能直接通过"OR"连接。

g. 如果"(+)"作用的列是和一个常量的比较关系， 那么这个表达式会成为join条件的一部分。

h. 同一个表不能对应多个外表。

i. "(+)"只能出现"比较表达式"，"NOT表达式"，“ANY表达式”，“ALL表达式”，“IN表达式”，“NULLIF表达式”，“IS DISTINCT FROM表达式”，“IS OF”表达式。"(+)"不能出现在其他类型表达式中，并且这些表达式中不允许出现通过“AND”和“OR”连接的表达式。

j. "(+)"只能转化为左外连接或者右外连接，不能转化为全连接，即不能在一个表达式的两个表上同时指定"(+)"

须知

对于WHERE子句的LIKE操作符，当LIKE中要查询特殊字符“%”、“_”、“\”的时候需要使用反斜杠“\”来进行转义。

示例：

```
CREATE TABLE tt01 (id int,content varchar(50));

INSERT INTO tt01 values (1,'Jack say ''hello''');
INSERT INTO tt01 values (2,'Rose do 50%');
INSERT INTO tt01 values (3,'Lilei say ''world''');
INSERT INTO tt01 values (4,'Hanmei do 100%');

SELECT * FROM tt01;
 id |      content
----+-------------------
  3 | Lilei say 'world'
  4 | Hanmei do 100%
  1 | Jack say 'hello'
  2 | Rose do 50%
(4 rows)

SELECT * FROM tt01 WHERE content like '%''he%';
 id |     content
----+------------------
  1 | Jack say 'hello'
(1 row)

SELECT * FROM tt01 WHERE content like '%50\%%';
 id |   content
----+-------------
  2 | Rose do 50%
(1 row)
```

- **GROUP BY子句**

  将查询结果按某一列或多列的值分组，值相等的为一组。

  – ROLLUP ({ expression | (expression [, ...]) } [, ...])

    ROLLUP是计算一个有序的分组列在GROUP BY中指定的标准聚集值，然后从右到左进一步创建高层次的部分和，最后创建了累积和。一个分组能够看做一系列的分组集。例如：

    ```
    GROUP BY ROLLUP (a,b,c)
    ```

    等价于：

    ```
    GROUP BY GROUPING SETS((a,b,c), (a,b), (a), ( ))
    ```

    ROLLUP子句中的元素可以是单独的字段或表达式，也可以是使用括号包含的列表。如果是括号中的列表，产生分组集时它们必须作为一个整体。例如：


```
GROUP BY ROLLUP ((a,b), (c,d))
```

等价于：

```
GROUPING SETS ((a,b,c,d), (a,b), (c,d ), ( ))
```

- CUBE ({ expression | (expression [, ...]) } [, ...])

  CUBE是自动对group by子句中列出的字段进行分组汇总，结果集将包含维度列中各值的所有可能组合，以及与这些维度值组合相匹配的基础行中的聚合值。它会为每个分组返回一行汇总信息， 用户可以使用CUBE来产生交叉表值。比如，在CUBE子句中给出三个表达式（n = 3），运算结果为$2^n = 2^3 = 8$组。 以n个表达式的值分组的行称为常规行，其余的行称为超级聚集行。例如：

  ```
  GROUP BY CUBE (a,b,c)
  ```

  等价于：

  ```
  GROUP BY GROUPING SETS((a,b,c), (a,b), (a,c), (b,c), (a), (b), (c), ( ))
  ```

  CUBE子句中的元素可以是单独的字段或表达式，也可以是使用括号包含的列表。如果是括号中的列表，产生分组集时它们必须作为一个整体。例如：

  ```
  GROUP BY CUBE (a, (b, c), d)
  ```

  等价于：

  ```
  GROUP BY GROUPING SETS ((a,b,c,d), (a,b,c), (a), ( ))
  ```

- GROUPING SETS (grouping_element [, ...])

  GROUPING SETS子句是GROUP BY子句的进一步扩展，它可以使用户指定多个GROUP BY选项。选项用于定义分组集，每个分组集都需要包含在单独的括号中，空白的括号（()）表示将所有数据当作一个组处理。 这样做可以通过裁剪用户不需要的数据组来提高效率。 用户可以根据需要指定所需的数据组进行查询。

须知

如果SELECT列表的表达式中引用了那些没有分组的字段，则会报错，除非使用了聚集函数，因为对于未分组的字段，可能返回多个数值。

- **HAVING子句**

  与GROUP BY子句配合用来选择特殊的组。HAVING子句将组的一些属性与一个常数值比较，只有满足HAVING子句中的逻辑表达式的组才会被提取出来。

- **WINDOW子句**

  一般形式为WINDOW window_name AS (window_definition) [， ...]，window_name是可以被随后的窗口定义所引用的名称，window_definition可以是以下的形式：

  [existing_window_name]

  [PARTITION BY expression [, ...]]

  [ORDER BY expression [ASC | DESC | USING operator] [NULLS { FIRST | LAST }] [, ...]]

  [frame_clause]

  frame_clause为窗函数定义一个窗口框架window frame，窗函数（并非所有）依赖于框架，window frame是当前查询行的一组相关行。frame_clause可以是以下的形式：

  [RANGE | ROWS] frame_start

[RANGE | ROWS] BETWEEN frame_start AND frame_end

frame_start和frame_end可以是：

UNBOUNDED PRECEDING

value PRECEDING（RANGE不支持）

CURRENT ROW

value FOLLOWING（RANGE不支持）

UNBOUNDED FOLLOWING

须知

对列存表的查询目前只支持row_number窗口函数，不支持frame_clause。

- **UNION子句**

  UNION计算多个SELECT语句返回行集合的并集。

  UNION子句有如下约束条件：

  - 除非声明了ALL子句，否则缺省的UNION结果不包含重复的行。
  - 同一个SELECT语句中的多个UNION操作符是从左向右计算的，除非用圆括弧进行了标识。
  - FOR UPDATE不能在UNION的结果或输入中声明。

  一般表达式：

  select_statement UNION [ALL] select_statement

  - select_statement可以是任何没有ORDER BY、LIMIT、FOR UPDATE子句的SELECT语句。
  - 如果用圆括弧包围，ORDER BY和LIMIT可以附着在子表达式里。

- **INTERSECT子句**

  INTERSECT计算多个SELECT语句返回行集合的交集，不含重复的记录。

  INTERSECT子句有如下约束条件：

  - 同一个SELECT语句中的多个INTERSECT操作符是从左向右计算的，除非用圆括弧进行了标识。
  - 当对多个SELECT语句的执行结果进行UNION和INTERSECT操作的时候，会优先处理INTERSECT。

  一般形式：

  select_statement INTERSECT select_statement

  select_statement可以是任何没有FOR UPDATE子句的SELECT语句。

- **EXCEPT子句**

  EXCEPT子句有如下的通用形式：

  select_statement EXCEPT [ALL] select_statement

  select_statement是任何没有FOR UPDATE子句的SELECT表达式。

  EXCEPT操作符计算存在于左边SELECT语句的输出而不存在于右边SELECT语句输出的行。

  EXCEPT的结果不包含任何重复的行，除非声明了ALL选项。使用ALL时，一个在左边表中有m个重复而在右边表中有n个重复的行将在结果中出现max(m-n,0) 次。

除非用圆括弧指明顺序，否则同一个SELECT语句中的多个EXCEPT操作符是从左向右计算的。EXCEPT和UNION的绑定级别相同。

目前，不能给EXCEPT的结果或者任何EXCEPT的输入声明FOR UPDATE子句。

- **MINUS子句**

  与EXCEPT子句具有相同的功能和用法。

- **ORDER BY子句**

  对SELECT语句检索得到的数据进行升序或降序排序。对于ORDER BY表达式中包含多列的情况：

  - 首先根据最左边的列进行排序，如果这一列的值相同，则根据下一个表达式进行比较，以此类推。
  - 如果对于所有声明的表达式都相同，则按随机顺序返回。
  - ORDER BY中排序的列必须包括在SELECT语句所检索的结果集的列中。

  须知

  - 如果未指定ORDER BY，则按数据库系统最快生成的顺序返回。
  - 可以选择在 ORDER BY子句中的任何表达式之后添加关键字ASC（升序）或DESC（降序）。如果未指定，则默认使用ASC。
  - 如果要支持中文拼音排序和不区分大小写排序，需要在初始化数据库时指定编码格式为UTF-8或GBK。 命令如下:

    initdb –E UTF8 –D ../data –locale=zh_CN.UTF-8或initdb –E GBK –D ../data –locale=zh_CN.GBK。

- **[{ [LIMIT { count | ALL }] [OFFSET start [ROW | ROWS]] } | { LIMIT start, { count | ALL } }]**

  LIMIT子句由两个独立的Limit子句、Offset子句和一个多参Limit子句构成：

  LIMIT { count | ALL }

  OFFSET start [ROW | ROWS]

  LIMIT start, { count | ALL }

  其中，count声明返回的最大行数，而start声明开始返回行之前忽略的行数。如果这两个参数都指定了，会在开始计算count个返回行之前先跳过start行。多参Limit子句不可和单参的Limit子句或Offset子句共同出现。

- **FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY**

  如果不指定count，默认值为1，FETCH子句限定返回查询结果从第一行开始的总行数。

- **FOR UPDATE子句**

  FOR UPDATE子句将对SELECT检索出来的行进行加锁。这样避免它们在当前事务结束前被其他事务修改或者删除，即其他企图UPDATE、 DELETE、 SELECT FOR UPDATE这些行的事务将被阻塞，直到当前事务结束。

  为了避免操作等待其他事务提交，可使用NOWAIT选项，如果被选择的行不能立即被锁住，执行SELECT FOR UPDATE NOWAIT将会立即汇报一个错误，而不是等待。

  FOR SHARE的行为类似，只是它在每个检索出来的行上要求一个共享锁，而不是一个排他锁。一个共享锁阻塞其它事务执行UPDATE、DELETE、SELECT，不阻塞SELECT FOR SHARE。

如果在FOR UPDATE或FOR SHARE中明确指定了表名字，则只有这些指定的表被锁定，其他在SELECT中使用的表将不会被锁定。否则，将锁定该命令中所有使用的表。

如果FOR UPDATE或FOR SHARE应用于一个视图或者子查询，它同样将锁定所有该视图或子查询中使用到的表。

多个FOR UPDATE和FOR SHARE子句可以用于为不同的表指定不同的锁定模式。

如果一个表中同时出现（或隐含同时出现）在FOR UPDATE和FOR SHARE子句中，则按照FOR UPDATE处理。类似的，如果影响一个表的任意子句中出现了NOWAIT，该表将按照NOWAIT处理。

须知

- 对于for update/share，执行计划不能下推的SQL，直接返回报错信息；对于执行计划可以下推的，下推到DN执行。
- 对列存表的查询不支持for update/share。

- **NLS_SORT**

  指定某字段按照特殊方式排序。目前仅支持中文拼音格式排序和不区分大小写排序。

  取值范围：

  - SCHINESE_PINYIN_M，按照中文拼音排序（目前只支持GBK字符集内的一级汉字排序）。如果要支持此排序方式，在创建数据库时需要指定编码格式为“GBK”，否则排序无效。
  - generic_m_ci，不区分大小写排序。

- **PARTITION子句**

  查询某个分区表中相应分区的数据。

示例

- WITH子句

  先通过子查询得到一张临时表temp_t，然后查询表temp_t中的所有数据：

  ```
  WITH temp_t(name,isdba) AS (SELECT usename,usesuper FROM pg_user) SELECT * FROM temp_t;
  ```

  为名为temp_t的with_query显示指定MATERIALIZED，然后查询表temp_t中的所有数据：

  ```
  WITH temp_t(name,isdba) AS MATERIALIZED (SELECT usename,usesuper FROM pg_user) SELECT * FROM temp_t;
  ```

  为名为temp_t的with_query显示指定NOT MATERIALIZED，然后查询表temp_t中的所有数据：

  ```
  WITH temp_t(name,isdba) AS NOT MATERIALIZED (SELECT usename,usesuper FROM pg_user)
   SELECT * FROM temp_t t1 WHERE name LIKE 'A%'
   UNION ALL
   SELECT * FROM temp_t t2 WHERE name LIKE 'B%';
  ```

- DISTINCT示例

  查询tpcds.reason表的所有r_reason_sk记录，且去除重复：

  ```
  CREATE SCHEMA tpcds;
  DROP TABLE IF EXISTS tpcds.reason;
  CREATE TABLE tpcds.reason(r_reason_sk integer not null,r_reason_id char(16) not null,r_reason_desc char(100));
  SELECT DISTINCT(r_reason_sk) FROM tpcds.reason;
  ```

- LIMIT子句示例

  获取表中第一条记录：

  ```
  SELECT * FROM tpcds.reason LIMIT 1;
  ```

  获取表中第三条记录：

  ```
  SELECT * FROM tpcds.reason LIMIT 1 OFFSET 2;
  ```

  获取表中前两条记录：

  ```
  SELECT * FROM tpcds.reason LIMIT 2;
  ```

- ORDER BY子句示例

  查询所有记录，且按字母升序排列：

  ```
  SELECT r_reason_desc FROM tpcds.reason ORDER BY r_reason_desc;
  ```

- SELECT列表示例

  通过表别名，从pg_user和pg_user_status这两张表中获取数据：

  ```
  SELECT a.usename,b.locktime FROM pg_user a,pg_user_status b WHERE a.usesysid=b.roloid;
  ```

- FULL JOIN子句示例

  将pg_user和pg_user_status这两张表的数据进行全连接显示，即数据的合集：

  ```
  SELECT a.usename,b.locktime,a.usesuper FROM pg_user a FULL JOIN pg_user_status b on
  a.usesysid=b.roloid;
  ```

- GROUP BY子句示例

  创建销售数据表sale：

  ```
  CREATE TABLE sales (
      item VARCHAR(10),
      year VARCHAR(4),
      quantity INT
  );

  INSERT INTO sales VALUES('apple', '2018', 800);
  INSERT INTO sales VALUES('apple', '2018', 1000);
  INSERT INTO sales VALUES('banana', '2018', 500);
  INSERT INTO sales VALUES('banana', '2018', 600);
  INSERT INTO sales VALUES('apple', '2019', 1200);
  INSERT INTO sales VALUES('banana', '2019', 1800);
  ```

  按照产品和年度的组合进行分组：

  ```
  SELECT item, year, SUM(quantity) FROM sales GROUP BY item, year;
  ```

  使用GROUPING SETS指定自定义的分组集，对可能出现的各种结果进行查询。以下查询返回按照产品和年度组合进行统计的销量小计，加上按照产品进行统计的销量合计，再加上按照年度进行统计的销量合计以及所有销量的总计：

  ```
  SELECT coalesce(item, '所有产品') AS "产品",
        coalesce(year, '所有年度') AS "年度",
        SUM(quantity) as "销量"
   FROM sales
   GROUP BY GROUPING SETS (
        (item, year),
        (item),
        (year),
        ()
       )
   ORDER BY item,year;
  ```

  随着分组字段的增加，通过GROUPING SETS列出所有可能的分组方式会显得比较麻烦，此时可使用简写形式的GROUPING SETS CUBE。

  ```
  SELECT coalesce(item, '所有产品') AS "产品",
        coalesce(year, '所有年度') AS "年度",
        SUM(quantity) as "销量"
   FROM sales
   GROUP BY CUBE (item,year)
   ORDER BY item,year;
  ```


GROUPING SETS ROLLUP则是用于生成按照层级进行汇总的结果。以下查询返回按照产品和年度组合进行统计的销量小计，加上按照产品进行统计的销量合计，再加上所有销量的总计：

```
SELECT coalesce(item, '所有产品') AS "产品",
       coalesce(year, '所有年度') AS "年度",
       SUM(quantity) as "销量"
 FROM sales
GROUP BY ROLLUP (item,year)
ORDER BY item,year;
```

- UNION子句示例

  将表tpcds.reason里r_reason_desc字段中的内容以W开头和以N开头的进行合并：

```
SELECT r_reason_sk, tpcds.reason.r_reason_desc
    FROM tpcds.reason
    WHERE tpcds.reason.r_reason_desc LIKE 'W%'
UNION
SELECT r_reason_sk, tpcds.reason.r_reason_desc
    FROM tpcds.reason
    WHERE tpcds.reason.r_reason_desc LIKE 'N%';
```

- NLS_SORT子句示例

  中文拼音排序：

```
CREATE TABLE stu_pinyin_info (id bigint, name text) DISTRIBUTE BY REPLICATION;
INSERT INTO stu_pinyin_info VALUES (1, '雷锋'),(2, '石传祥');
SELECT * FROM stu_pinyin_info ORDER BY NLSSORT (name, 'NLS_SORT = SCHINESE_PINYIN_M' );
```

  不区分大小写排序：

```
CREATE TABLE stu_icase_info (id bigint, name text) DISTRIBUTE BY REPLICATION;
INSERT INTO stu_icase_info VALUES (1, 'aaaa'),(2, 'AAAA');
SELECT * FROM stu_icase_info ORDER BY NLSSORT (name, 'NLS_SORT = generic_m_ci');
```

- 查询分区表示例

  创建分区表tpcds.reason_p，并插入数据，再从tpcds.reason_p的表分区P_05_BEFORE中获取数据。

```
CREATE TABLE tpcds.reason_p
(
  r_reason_sk integer,
  r_reason_id character(16),
  r_reason_desc character(100)
)
PARTITION BY RANGE (r_reason_sk)
(
  partition P_05_BEFORE values less than (05),
  partition P_15 values less than (15),
  partition P_25 values less than (25),
  partition P_35 values less than (35),
  partition P_45_AFTER values less than (MAXVALUE)
);

INSERT INTO tpcds.reason_p values(3,'AAAAAAAABAAAAAAA','reason 1'),
(10,'AAAAAAAABAAAAAAA','reason 2'),(4,'AAAAAAAABAAAAAAA','reason 3'),
(10,'AAAAAAAABAAAAAAA','reason 4'),(10,'AAAAAAAABAAAAAAA','reason 5'),
(20,'AAAAAAAACAAAAAAA','reason 6'),(30,'AAAAAAAACAAAAAAA','reason 7');
```

  查询指定分区：

```
SELECT * FROM tpcds.reason_p PARTITION (P_05_BEFORE);
```

  查询分区P_15的行数：

```
SELECT count(*) FROM tpcds.reason_p PARTITION (P_15);
```

- HAVING子句示例

  按r_reason_id分组统计tpcds.reason_p表中的记录，并只显示r_reason_id个数大于2的信息：

```
SELECT COUNT(*) c,r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id HAVING c>2;
```


- IN子句示例

  按r_reason_id分组统计tpcds.reason_p表中的r_reason_id个数，并只显示r_reason_id值为 AAAAAAAABAAAAAAA或AAAAAAAADAAAAAAA的个数：

```
SELECT COUNT(*),r_reason_id FROM tpcds.reason_p GROUP BY r_reason_id HAVING r_reason_id IN('AAAAAAAABAAAAAAA','AAAAAAAADAAAAAAA');
```

- INTERSECT子句示例

  查询r_reason_id等于AAAAAAAABAAAAAAA，并且r_reason_sk小于5的信息：

```
SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA' INTERSECT SELECT * FROM tpcds.reason_p WHERE r_reason_sk<5;
```

- EXCEPT子句示例

  查询r_reason_id等于AAAAAAAABAAAAAAA，并且去除r_reason_sk小于4的信息：

```
SELECT * FROM tpcds.reason_p WHERE r_reason_id='AAAAAAAABAAAAAAA' EXCEPT SELECT * FROM tpcds.reason_p WHERE r_reason_sk<4;
```

- WHERE子句示例

  通过在where子句中指定"(+)"来实现左连接：

```
DROP TABLE IF EXISTS store_returns;
CREATE TABLE store_returns
(
   sr_item_sk    BIGINT ,
   sr_customer_sk        VARCHAR(25)
);

DROP TABLE IF EXISTS customer;
CREATE TABLE customer
(
   C_CUSTKEY     BIGINT NOT NULL CONSTRAINT C_CUSTKEY_pk PRIMARY KEY  ,
   C_NAME        VARCHAR(25)  ,
   C_CUSTOMER_SK VARCHAR(25)  ,
   C_CUSTOMER_ID VARCHAR(40)  ,
   C_NATIONKEY   INT          ,
   C_PHONE       CHAR(15)     ,
   C_ACCTBAL     DECIMAL(15,2)
);

SELECT t1.sr_item_sk ,t2.c_customer_id FROM store_returns t1, customer t2 WHERE t1.sr_customer_sk = t2.c_customer_sk(+) order by 1 desc limit 1;
```

  通过在where子句中指定"(+)"来实现右连接：

```
SELECT t1.sr_item_sk ,t2.c_customer_id FROM store_returns t1, customer t2 WHERE t1.sr_customer_sk(+)  = t2.c_customer_sk order by 1 desc limit 1;
```

  通过在where子句中指定"(+)"来实现左连接，并且增加连接条件：

```
SELECT t1.sr_item_sk ,t2.c_customer_id FROM store_returns t1, customer t2 WHERE t1.sr_customer_sk = t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1 order by 1  limit 1;
```

  不支持在where子句中指定"(+)"的同时使用内层嵌套AND/OR的表达式：

```
SELECT t1.sr_item_sk ,t2.c_customer_id FROM store_returns t1, customer t2 WHERE not(t1.sr_customer_sk  = t2.c_customer_sk(+) and t2.c_customer_sk(+) < 1);
ERROR:  Operator "(+)" can not be used in nesting expression.
LINE 1: ...tomer_id from store_returns t1, customer t2 where not(t1.sr_...
```

  where子句在表达式宏指定"(+)"会报错：

```
SELECT t1.sr_item_sk ,t2.c_customer_id FROM store_returns t1, customer t2 WHERE (t1.sr_customer_sk  = t2.c_customer_sk(+))::bool;
ERROR:  Operator "(+)" can only be used in common expression.
```

  where子句在表达式的两边都指定"(+)"会报错：

```
SELECT t1.sr_item_sk ,t2.c_customer_id FROM store_returns t1, customer t2 WHERE t1.sr_customer_sk(+)  = t2.c_customer_sk(+);
ERROR:  Operator "(+)" can't be specified on more than one relation in one join condition
HINT:  "t1", "t2"...are specified Operator "(+)" in one condition.
```


15.3 SELECT INTO

功能描述

SELECT INTO用于根据查询结果创建一个新表，并且将查询到的数据插入到新表中。

数据并不返回给客户端，这一点和普通的SELECT不同。新表的字段具有和SELECT的输出字段相同的名字和数据类型。

注意事项

CREATE TABLE AS的作用和SELECT INTO类似，且提供了SELECT INTO所提供功能的超集。建议使用CREATE TABLE AS语法替代SELECT INTO，因为SELECT INTO不能在存储过程中使用并且SELECT INTO 字段名，不支持接收空行。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    { * | {expression [ [ AS ] output_name ]} [, ...] }
    INTO [ UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW {window_name AS ( window_definition )} [, ...] ]
    [ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
    [ ORDER BY {expression [ [ ASC | DESC | USING operator ] | nlssort_expression_clause ] [ NULLS { FIRST |
LAST } ]} [, ...] ]
    [ { [ LIMIT { count | ALL } ] [ OFFSET start [ ROW | ROWS ] ] } | { LIMIT start, { count | ALL } } ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ]} [...] ];
```

参数说明

INTO [UNLOGGED] [TABLE] new_table

UNLOGGED指定表为非日志表。写入非日志表中的数据不会写入到预写日志中，这样就会比普通表快很多。但是，非日志表也是不安全的，在冲突或异常关机后会被自动删截。非日志表中的内容也不会被复制到备用服务器中。在该类表中创建的索引也不会被自动记录。

new_table指定新建表的名字。

说明

SELECT INTO的其它参数可参考SELECT的**参数说明**。

示例

将reason_t表中TABLE_SK小于3的值加入到新建表中。

```
CREATE TABLE IF NOT EXISTS reason_t
(
    TABLE_SK         INTEGER              ,
    TABLE_ID         VARCHAR(20)          ,
    TABLE_NA          VARCHAR(20)
);
```

```
INSERT INTO reason_t VALUES (1, 'S01', 'StudentA'),(2, 'T01', 'TeacherA'),(3, 'T02', 'TeacherB'),(3, 'S02',
'StudentB');

SELECT * INTO reason_t_bck FROM reason_t WHERE TABLE_SK < 3;
```

相关链接

SELECT

16 TCL 语法

16.1 TCL 语法一览表

TCL（Transaction Control Language事务控制语言 ），用来控制数据库操纵事务发生的时间及效果，对数据库实行监视等。

提交

GaussDB(DWS)通过COMMIT或者END可完成提交事务的功能。请参考**COMMIT | END**。

设置保存点

GaussDB(DWS)用于在当前事务里建立一个新的保存点。请参考**SAVEPOINT**。

回滚

GaussDB(DWS)回滚当前事务状态回到上次最后提交的状态。请参考**ROLLBACK**。

16.2 ABORT

功能描述

回滚当前事务并且撤销所有当前事务中所做的更改。

作用等同于**ROLLBACK**，早期SQL有用ABORT，现在推荐使用ROLLBACK。

注意事项

在事务外部执行ABORT语句不会影响事务的执行，但是会产生一个警告信息。

语法格式

```
ABORT [ WORK | TRANSACTION ] ;
```

参数说明

WORK | TRANSACTION

可选关键字，除了增加可读性没有其他任何作用。

示例

终止事务，执行的更新操作会被撤销掉：

```
ABORT;
```

相关链接

SET TRANSACTION，**COMMIT | END**，**ROLLBACK**

16.3 BEGIN

功能描述

BEGIN可以用于开始一个匿名块，也可以用于开始一个事务。本节描述用BEGIN开始匿名块的语法，BEGIN开始事务的语法见**START TRANSACTION**。

匿名块是能够动态地创建和执行过程代码的结构，而不需要以持久化的方式将代码作为数据库对象储存在数据库中。

注意事项

无。

语法格式

- 开启匿名块

```
[DECLARE [declare_statements]]
BEGIN
execution_statements
END;
/
```

- 开启事务

```
BEGIN [ WORK | TRANSACTION ]
 [
   {
     ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE
READ }
     | { READ WRITE | READ ONLY }
    } [, ...]
 ];
```

参数说明

- **declare_statements**

  声明变量，包括变量名和变量类型，如“sales_cnt int”。

- **execution_statements**

  匿名块中要执行的语句。

  取值范围：已存在的函数名称。

示例

- 开始事务块：
  ```
  BEGIN;
  ```
- 要以可重复读隔离级别开始事务块：
  ```
  BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
  ```
- 使用匿名块输出字符串：
  ```
  BEGIN
  dbms_output.put_line('Hello');
  END;
  /
  ```

相关链接

START TRANSACTION

16.4 CHECKPOINT

功能描述

检查点（CHECKPOINT）是一个事务日志中的点，所有数据文件都在该点被更新以反映日志中的信息，所有数据文件都将被刷新到磁盘。

设置事务日志检查点。预写式日志（WAL）缺省时在事务日志中每隔一段时间放置一个检查点。可以设置相关运行时参数（checkpoint_segments和checkpoint_timeout）来调整这个原子化检查点的间隔。

注意事项

- 只有系统管理员可以调用CHECKPOINT。
- CHECKPOINT强制立即进行检查，而不是等到下一次调度时的检查点。

语法格式

```
CHECKPOINT;
```

参数说明

无。

示例

设置检查点：

```
CHECKPOINT;
```

16.5 COMMIT | END

功能描述

通过COMMIT或者END可完成提交事务的功能，即提交事务的所有操作。

注意事项

执行COMMIT这个命令的时候，命令执行者必须是该事务的创建者或系统管理员，且创建和提交操作可以不在同一个会话中。

语法格式

```
{ COMMIT | END } [ WORK | TRANSACTION ] ;
```

参数说明

- **COMMIT | END**

  提交当前事务，让所有当前事务的更改为其他事务可见。

- **WORK | TRANSACTION**

  可选关键字，除了增加可读性没有其他任何作用。

示例

提交事务，让所有更改永久化：

```
COMMIT;
```

相关链接

ROLLBACK

16.6 COMMIT PREPARED

功能描述

提交一个早先为两阶段提交准备好的事务。

注意事项

- 该功能仅在维护模式(GUC参数xc_maintenance_mode为on时)下可用。该模式谨慎打开，一般供维护人员排查问题使用，一般用户不应使用该模式。
- 命令执行者必须是该事务的创建者或系统管理员，且创建和提交操作可以不在同一个会话中。
- 事务功能由数据库自动维护，不应显式使用事务功能。

语法格式

```
COMMIT PREPARED transaction_id ;
COMMIT PREPARED transaction_id WITH CSN;
```

参数说明

- **transaction_id**

  待提交事务的标识符。它不能和任何当前预备事务已经使用了的标识符同名。

- **CSN(commit sequence number)**

  待提交事务的序列号。它是一个64位递增无符号数。

相关链接

PREPARE TRANSACTION，**ROLLBACK PREPARED**。

16.7 PREPARE TRANSACTION

功能描述

为两阶段提交准备当前事务。

在这个命令之后，该事务不再与当前会话关联。相反，它的状态完全保存在磁盘上，并且它有很高的可能性被提交成功，即使是在请求提交之前数据库发生了崩溃也如此。

一旦被准备好，事务稍后就可以用**COMMIT PREPARED**或 **ROLLBACK PREPARED**命令分别进行提交或者回滚。这些命令可以从任何会话中发出，而不仅仅是执行原始事务的会话。

从发出命令的会话的角度来看，PREPARE TRANSACTION不同于ROLLBACK：在执行它之后，就不再有活跃的当前事务了，并且预备事务的效果也不再可见 (如果该事务被提交，其效果会重新变得可见)。

如果PREPARE TRANSACTION因为某些原因失败，那么它就会变成一个ROLLBACK，当前事务被取消。

注意事项

- 事务功能由数据库自动维护，不应显式使用事务功能。
- 在运行PREPARE TRANSACTION命令时，必须在postgresql.conf配置文件中增大max_prepared_transactions的数值。建议至少将其设置为等于max_connections，这样每个会话都可以有一个等待中的预备事务。

语法格式

```
PREPARE TRANSACTION transaction_id;
```

参数说明

transaction_id

待提交事务的标识符，用于后面在COMMIT PREPARED或ROLLBACK PREPARED的时候标识这个事务。它不能和任何当前预备事务已经使用了的标识符同名。

取值范围：标识符必须以字符串文本的方式书写，并且必须小于200字节长。

相关链接

COMMIT PREPARED，**ROLLBACK PREPARED**

16.8 SAVEPOINT

功能描述

SAVEPOINT用于在当前事务里建立一个新的保存点。

保存点是事务中的一个特殊记号，它允许将那些在它建立后执行的命令全部回滚，把事务的状态恢复到保存点所在的时刻。

注意事项

- 使用ROLLBACK TO SAVEPOINT回滚到一个保存点。使用RELEASE SAVEPOINT删除一个保存点，但是保留该保存点建立后执行的命令的效果。
- 保存点只能在一个事务块里面建立。在一个事务里面可以定义多个保存点。
- 函数、匿名块和存储过程中不支持使用SAVEPOINT语法。
- 由于节点故障或者通信故障引起的分布式节点线程或进程退出导致的报错，以及由于COPY FROM操作中源数据与目标表的表结构不一致导致的报错，均不能正常回滚到保存点之前，而是整个事务回滚。
- SQL标准要求，使用savepoint建立一个同名保存点时，需要自动删除前面那个同名保存点。在GaussDB(DWS)数据库里，将保留旧的保存点，但是在回滚或者释放的时候，只使用最近的那个。释放了新的保存点将导致旧的再次成为ROLLBACK TO SAVEPOINT和RELEASE SAVEPOINT可以访问的保存点。除此之外，SAVEPOINT是完全符合SQL标准的。

语法格式

```
SAVEPOINT savepoint_name;
```

参数说明

savepoint_name

新建保存点的名字。

示例

- 建立一个保存点，然后撤销建立保存点后执行的所有命令的效果：

```
START TRANSACTION;
CREATE TABLE IF NOT EXISTS table1 (a int,b int);
INSERT INTO table1 VALUES (1);
SAVEPOINT my_savepoint;
INSERT INTO table1 VALUES (2);
ROLLBACK TO SAVEPOINT my_savepoint;
INSERT INTO table1 VALUES (3);
COMMIT;
```

查询表的内容，会同时看到1和3，不能看到2，因为2被回滚。

- 建立并随后销毁一个保存点：

```
START TRANSACTION;
INSERT INTO table1 VALUES (3);
SAVEPOINT my_savepoint;
INSERT INTO table1 VALUES (4);
RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

查询表的内容，会同时看到3和4。

相关链接

RELEASE SAVEPOINT，**ROLLBACK TO SAVEPOINT**

16.9 SET TRANSACTION

功能描述

为当前事务设置特性。它对后面的事务没有影响。事务特性包括事务隔离级别、事务访问模式(读/写或者只读)。

注意事项

无。

语法格式

设置事务的隔离级别、读写模式。

```
{ SET [ LOCAL ] TRANSACTION|SET SESSION CHARACTERISTICS AS TRANSACTION }
 { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
 | { READ WRITE | READ ONLY } } [, ...]
```

参数说明

- **LOCAL**

  声明该命令只在当前事务中有效。

- **SESSION**

  声明这个命令只对当前会话起作用。

  取值范围：字符串，要符合标识符的命名规范。

- **ISOLATION LEVEL**

  指定事务隔离级别，该参数决定当一个事务中存在其他并发运行事务时能够看到什么数据。

  说明

  - 在事务中第一个数据修改语句（INSERT，DELETE，UPDATE，FETCH，COPY）执行之后，事务隔离级别就不能再次设置。

  取值范围：

  - READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。这是缺省值。
  - READ UNCOMMITTED：读未提交隔离级别，GaussDB(DWS)不支持READ UNCOMMITTED，如果设置了READ UNCOMMITTED，实际上使用的是READ COMMITTED。
  - REPEATABLE READ：可重复读隔离级别，仅仅能看到事务开始之前提交的数据，不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。
  - SERIALIZABLE：事务可序列化，GaussDB(DWS)不支持SERIALIZABLE，如果设置了SERIALIZABLE，实际上使用的是REPEATABLE READ。

- **READ WRITE | READ ONLY**

  指定事务访问模式（读/写或者只读）。

示例

设置当前事务的隔离级别为READ COMMITTED，访问模式为READ ONLY：

```
START TRANSACTION;
SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED READ ONLY;
COMMIT;
```

16.10 START TRANSACTION

功能描述

通过START TRANSACTION启动事务。如果声明了隔离级别、读写模式，那么新事务就使用这些特性，类似执行了**SET TRANSACTION**。

注意事项

无。

语法格式

格式一：START TRANSACTION格式

```
START TRANSACTION
 [
  {
    ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
    | { READ WRITE | READ ONLY }
  } [, ...]
 ];
```

格式二：BEGIN格式

```
BEGIN [ WORK | TRANSACTION ]
 [
  {
    ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
    | { READ WRITE | READ ONLY }
  } [, ...]
 ];
```

参数说明

- **WORK | TRANSACTION**

  BEGIN格式中的可选关键字，没有实际作用。

- **ISOLATION LEVEL**

  指定事务隔离级别，它决定当一个事务中存在其他并发运行事务时它能够看到什么数据。

  说明

  在事务中第一个数据修改语句（INSERT，DELETE，UPDATE，FETCH，COPY）执行之后，事务隔离级别就不能再次设置。

  取值范围：

  - READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。这是缺省值。

  - READ UNCOMMITTED：读未提交隔离级别，GaussDB(DWS)不支持READ UNCOMMITTED，如果设置了READ UNCOMMITTED，实际上使用的是READ COMMITTED。
  - REPEATABLE READ： 可重复读隔离级别，仅仅看到事务开始之前提交的数据，它不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。
  - SERIALIZABLE：事务可序列化，GaussDB(DWS)不支持SERIALIZABLE，如果设置了SERIALIZABLE，实际上使用的是REPEATABLE READ。
- **READ WRITE | READ ONLY**

  指定事务访问模式（读/写或者只读）。

示例

- 以默认方式启动事务：

```
START TRANSACTION;
CREATE TABLE IF NOT EXISTS table1 (a int,b int);
SELECT * FROM table1;
END;
```

- 以隔离级别为READ COMMITTED，读/写方式启动事务：

```
START TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
CREATE TABLE IF NOT EXISTS table1 (a int,b int);
SELECT * FROM table1;
COMMIT;
```

相关链接

COMMIT | END，**ROLLBACK**，**SET TRANSACTION**

16.11 ROLLBACK

功能描述

回滚当前事务并取消当前事务中的所有更新。

在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销，数据库状态回到事务开始时。

注意事项

如果不在一个事务内部发出ROLLBACK不会有问题，但是将抛出一个警告信息。

语法格式

```
ROLLBACK [ WORK | TRANSACTION ];
```

参数说明

WORK | TRANSACTION

可选关键字。除了增加可读性，没有任何其他作用。

示例

取消当前事务中的所有更改：


```
ROLLBACK;
```

相关链接

COMMIT | END

16.12 RELEASE SAVEPOINT

功能描述

RELEASE SAVEPOINT删除当前事务先前定义的保存点。

保存点删除后便无法再作为回滚点使用，除此之外没有其它用户可见的行为。删除保存点并不能撤销在保存点建立起来之后执行的命令的影响。要撤销那些命令可以使用ROLLBACK TO SAVEPOINT 。在不再需要的时候删除保存点可以令系统在事务结束之前提前回收一些资源。

RELEASE SAVEPOINT也删除所有在指定的保存点建立之后的所有保存点。

注意事项

- 不能RELEASE没有定义的保存点，语法上会报错。
- 如果事务在回滚状态，则不能释放保存点。
- 如果多个保存点拥有同样的名字，只释放最近定义的保存点。

语法格式

```
RELEASE [ SAVEPOINT ] savepoint_name;
```

参数说明

savepoint_name

要删除的保存点的名字

示例

建立并随后销毁一个保存点：

```
BEGIN;
    CREATE TABLE IF NOT EXISTS table1 (a int,b int);
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

相关链接

SAVEPOINT，**ROLLBACK TO SAVEPOINT**

16.13 ROLLBACK PREPARED

功能描述

取消一个先前为两阶段提交准备好的事务。

注意事项

- 该功能仅在维护模式(GUC参数xc_maintenance_mode为on时)下可用。该模式谨慎打开，一般供维护人员排查问题使用，一般用户不应使用该模式。
- 要想回滚一个预备事务，必须是最初发起事务的用户，或者是系统管理员。
- 事务功能由数据库自动维护，不应显式使用事务功能。

语法格式

```
ROLLBACK PREPARED transaction_id ;
```

参数说明

transaction_id

待提交事务的标识符。它不能和任何当前预备事务已经使用了的标识符同名。

相关链接

COMMIT PREPARED，**PREPARE TRANSACTION**。

16.14 ROLLBACK TO SAVEPOINT

功能描述

ROLLBACK TO SAVEPOINT用于回滚到保存点，隐含地删除所有在该保存点之后建立的保存点。

回滚所有指定保存点建立之后执行的命令。保存点仍然有效，并且需要时可以再次回滚到该点。

注意事项

- 不能回滚到未定义的保存点，语法上会报错。
- 在保存点方面，游标有一些非事务性的行为。任何在保存点里打开的游标都会在回滚掉这个保存点之后关闭。如果一个前面打开了的游标在保存点里面，并且游标被一个FETCH命令影响，而这个保存点稍后回滚了，那么这个游标的位置仍然在FETCH让它指向的位置(也就是FETCH不会被回滚)。关闭一个游标的行为也不会被回滚给撤销掉。如果一个游标的操作导致事务回滚，那么这个游标就会置于不可执行状态，所以，尽管一个事务可以用ROLLBACK TO SAVEPOINT重新恢复，但是游标不能再使用了。
- 使用ROLLBACK TO SAVEPOINT回滚到保存点。使用RELEASE SAVEPOINT删除保存点，但是保留该保存点建立后执行的命令的效果。

语法格式

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name;
```

参数说明

savepoint_name

回滚截至的保存点

示例

撤销 my_savepoint 建立之后执行的命令的影响：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

游标位置不受保存点回滚的影响：

```
BEGIN;
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;
SAVEPOINT foo;
FETCH 1 FROM foo;
 ?column?
----------
        1
ROLLBACK TO SAVEPOINT foo;
FETCH 1 FROM foo;
 ?column?
----------
        2
COMMIT;
```

相关链接

SAVEPOINT，**RELEASE SAVEPOINT**